Universitat de les Illes Balears

Facultat de Psicologia

Tesis doctoral

Análisis y Diseño Orientado a Objetos de un *Framework* para el Modelado Estadístico con MLG

Rafael Jiménez López

Directores:
Dr. Josep Maria Losilla Vidal
Dr. Alfonso L. Palmer Pol

Palma de Mallorca, 2003

Esta tesis ha sido realizada gracias a los proyectos BSO2001-2518, BSO2002-2513 y BSO2001-0369 concedidos por el Ministerio de Ciencia y Tecnología.

Agradecimientos

Quiero manifestar mi agradecimiento a mis maestros y directores de tesis, Dr. Josep Maria Losilla Vidal y Dr. Alfonso L. Palmer Pol, no sólo por su impecable labor de dirección sino también por sus continuas enseñanzas y apoyo en la elaboración de este trabajo. Quisiera también expresar mi gratitud a Jaume Vives, y a mis compañeros Alberto Sesé, Juan José Montaño, Berta Cajal y Noelia Llorens, por su respaldo personal y amistad incondicional. Finalmente, agradezco el apoyo sincero de mi familia y amigos. De todo corazón, muchas gracias a todos.

ÍNDICE

1. INTRODUCCIÓN Y OBJETIVOS

La motivación de esta tesis doctoral se centra en la necesidad de encontrar metodologías, procedimientos y herramientas enmarcadas en el campo de la ingeniería del software, que ayuden al profesional e investigador de la estadística en el análisis de los problemas a los que se enfrenta durante su labor cotidiana, al tiempo que guíen el proceso de diseño de soluciones informáticas concretas a dichos problemas.

Hoy en día es todavía habitual que un profesional de la estadística se vea limitado en su quehacer laboral por las prestaciones de los paquetes estadísticos convencionales, como SPSS, Minitab, etc., utilizando estas herramientas al nivel de usuario, cuando en realidad sus necesidades van más lejos, requiriendo un papel más activo y el empleo de entornos de desarrollo más avanzados y flexibles, como S-Plus, R, Matlab o ViSta.

Es precisamente en el campo de la investigación estadística desde donde se formulan la mayoría de demandas que requieren soluciones informatizadas especiales. Existen multitud de herramientas de simulación implementadas en diferentes lenguajes de programación, y el experto en estadística las utiliza para simular, por ejemplo, muestras aleatorias bajo una determinada distribución teórica, para estimar errores estándar, para probar la potencia de determinadas pruebas estadísticas, etc.

Yendo más allá de un uso exclusivo de funciones implementadas por otros, el experto se ve habitualmente en la necesidad de plantear el diseño de extensiones que amplíen esta funcionalidad. Este es el rol que debe asumir un investigador en estadística; además de nutrirse de las herramientas de simulación recogidas en distintos sistemas de análisis de datos, debe ser capaz de añadir nuevos elementos a dicho sistema para solucionar problemas específicos relacionados con su investigación.

Es en este sentido, que Ocaña y Sánchez (2003) afirman que una persona que investiga en estadística es, en mayor o menor grado, también un desarrollador informático. Chambers (2000) analiza esta cuestión y deduce que, en realidad, existe una gradación entre estos extremos de utilización de la informática en la estadística. Un usuario que inicialmente se limitó a realizar tareas elementales con un programa informático, puede verse en la necesidad de agruparlas de alguna manera para evitar la realización de tareas repetitivas, lo cual ya es un primer paso hacia la programación. Por su parte, un investigador seguramente

se verá en la necesidad de extender cierta funcionalidad del sistema informático que maneja, para cubrir soluciones que no habían sido todavía implementadas.

Chambers (2000) hace hincapié precisamente en la necesidad de que un usuario de software estadístico moderno no se limite a emplear los componentes ya implementados por otros, sino que dé un paso más y sea capaz de desarrollar modificaciones en dichos componentes o crear otros nuevos que resuelvan su problema específico. Para que este tránsito de simple usuario a programador sea viable se requiere trabajar en el contexto de herramientas estadísticas modernas. Chambers plantea en este sentido una lista de cinco condiciones básicas que debería cumplir toda herramienta informática empleada en Estadística:

1. Especificación fácil de tareas sencillas;
2. capacidad de refinamiento gradual de las tareas;
3. posibilidades ilimitadas de extensión mediante programación;
4. desarrollo de programas de alta calidad; y
5. posibilidad de integrar los resultados de los puntos 2 a 4 como nuevas herramientas informáticas.

En esta línea, es destacable la reciente publicación de la plataforma estadística *Omegahat* (Temple, 2000), que permite diseñar sistemas estadísticos orientados a objetos. Este nuevo entorno de desarrollo estadístico se basa en el lenguaje Java para la construcción y ampliación de sistemas, lenguaje éste de uso creciente por las empresas desarrolladoras de software por su gran flexibilidad, su rica implementación de los conceptos derivados del paradigma de Orientación a Objetos, y su facilidad para el desarrollo de herramientas de ejecución distribuida. De hecho, Omegahat, aunque todavía se encuentra en fase de evolución, supera a otros entornos de desarrollo como R o S-Plus en el cumplimiento de las condiciones planteadas por Chambers (2000), tal como se analiza en el apartado 3.3 de nuestro trabajo.

A partir de un entorno de programación con estas características, se puede plantear la construcción de una nueva herramienta o aplicación específica para un dominio de problemas concreto, procurando que ésta sea fácilmente extensible. Este tipo de aplicaciones se denominan *frameworks*. Los entornos de trabajo y de desarrollo orientados

a objetos están preparados para construir *frameworks* que permitan la reutilización del diseño y del código del sistema que implementan.

El desarrollo de sistemas completos o subsistemas concretos (*frameworks*) enmarcados en el contexto de un determinado dominio de la estadística supera normalmente el rol que adopta un investigador estadístico. Se habla en este caso de diseñador-desarrollador en el ámbito de la estadística. En este sentido, el propósito fundamental de este trabajo es llevar a cabo una revisión en profundidad de los procedimientos y herramientas necesarios para que un investigador adquiera la autonomía suficiente para diseñar (y opcionalmente implementar) extensiones en plataformas ya existentes –entornos que cubran las condiciones propuestas por Chambers.

La existencia de un procedimiento normalizado para el desarrollo de software –como es el caso del *Proceso Unificado* (UP) (Jacobson, Booch y Rumbaugh, 2000)– y la existencia de un lenguaje simbólico común –el *Lenguaje Unificado de Modelado* (UML) (Booch, Rumbaugh y Jacobson, 1999)– para expresar el análisis del problema y el diseño e implementación de la solución planteados durante el UP, facilitan en gran medida que el profesional de la estadística pueda actuar también como desarrollador informático, y no sólo como experto en el dominio del problema. Además, el UP, junto al lenguaje UML, se enmarca dentro del paradigma de la Orientación a Objetos (OO), que constituye una forma más fácil y natural de entender el desarrollo de *software*.

El camino a seguir, tanto en el marco teórico como en la parte empírica de este trabajo, pasa por destacar la importancia de contar con esta metodología para que un profesional de la estadística sea capaz de desarrollar, no sólo extensiones concretas, sino sistemas completos enmarcados en un dominio concreto del análisis de datos.

Por supuesto, la tarea de desarrollar software entraña a simple vista una gran dificultad, pues supone el dominio de un lenguaje de programación específico. Esta dificultad se suaviza cuando se analizan los problemas y se diseñan las soluciones bajo el paradigma de la OO y se siguen las pautas de desarrollo de software marcadas por el UP. En el apartado 2.3.2 se analizan las diferencias entre la perspectiva de programación estructurada (tradicionalmente empleada por los desarrolladores de software) y la perspectiva de programación orientada a objetos.

Pensar en términos de objetos es algo innato en todas las personas. De hecho, la capacidad de reconocer objetos físicos es una habilidad que los humanos aprenden en edades muy tempranas; así, cuando el niño desarrolla el *concepto de objeto* es capaz de buscar una pelota escondida que le habían mostrado previamente. A través del concepto de objeto, un niño llega a darse cuenta de que los objetos tienen una permanencia e identidad, además de un determinado comportamiento. Por supuesto, desde la perspectiva de la cognición humana, un objeto no sólo es algo tangible, puede ser algo que pueda comprenderse intelectualmente o algo hacia lo que se dirige un pensamiento o una acción. Como se analizará en su momento, esta capacidad innata de pensar en términos de objetos tiene su fiel reflejo en el análisis, diseño e implementación OO.

Por otra parte, el hecho de programar bajo el paradigma de la OO supone hacerlo a muy alto nivel, en el sentido de que es posible implementar la solución a nuestro problema, en general, con un mínimo de instrucciones. Esto es así debido a que no se parte de cero en dicha implementación, puesto que la mayoría de lenguajes de programación orientados a objetos incluyen un extenso repertorio de clases. En este contexto, la tarea del desarrollador consistirá normalmente en buscar aquellas clases cuyas propiedades y comportamiento satisfagan su necesidad, instanciando a partir de ellas los objetos de interés y estableciendo las interacciones entre dichos objetos para obtener el resultado esperado. Esta manera de proceder supone una forma de programar bastante intuitiva, puesto que fuerza a pensar en objetos que colaboran entre sí para producir un determinado resultado de valor. Realmente, es como interaccionan los objetos de nuestro entorno para conseguir cambios de estado en los sistemas que nos envuelven; un coche inicia su marcha gracias a la colaboración de los elementos (objetos) que componen el motor y el sistema de transmisión de energía a las ruedas.

Cuando se necesita que un objeto disponga de un comportamiento adicional, el desarrollador puede derivar una nueva clase que herede las características y operaciones de una clase *padre* (que incluya las operaciones y atributos de interés), y añadir a esta clase derivada la nueva operación. De esta manera, se dispone de una clase que permite instanciar un nuevo tipo de objeto (que incluye el comportamiento adicional). La *reusabilidad* es la principal ventaja del paradigma OO, que evita tener que “reinventar la rueda”.

Esta idea de la reusabilidad posee un gran atractivo en cualquier contexto de programación. Por supuesto, también en el contexto de la programación estadística; puede alentar a un profesional de la estadística a desarrollar soluciones concretas a sus necesidades de investigación, siempre y cuando se asuma esta forma de entender la programación. Otro de los objetivos implícitos de esta tesis es aportar argumentos válidos para que se produzca ese cambio de mentalidad. En este sentido, la programación en entornos de desarrollo integrados (EDI) orientados a objetos, que además proporcionen asistencia al investigador durante la sesión de trabajo (ayuda sintáctica, sugerencias, etc.), facilitaría ese acercamiento hacia el desarrollo de software estadístico.

Se ha de tener en cuenta que el UP es precisamente una parte esencial en la construcción de software, puesto que guía el proceso reduciendo riesgos en las decisiones de análisis, diseño e implementación. Por otro lado, el UML es fundamental como lenguaje de publicación de los artefactos generados durante el UP, puesto que permite expresar todos los esquemas y diagramas del sistema software de forma precisa y sintética. En esta línea, Losilla (2003) opina que en las publicaciones estadísticas que incluyen desarrollos informáticos, la práctica habitual consistente en la publicación únicamente del análisis en lenguaje natural (aunque incluya notación matemática), y/o de su implementación en algún lenguaje de programación específico, es insuficiente desde el punto de vista de la comunicación científica; de forma complementaria, dichas publicaciones deberían incluir al menos los modelos de diseño expresados en UML.

En este sentido, y siguiendo el planteamiento de Losilla (2003), la implementación de software estadístico siguiendo el proceso de modelado enmarcado en el UP, y la publicación de sus artefactos en UML, facilita la comprensión, revisión, replicación, mantenimiento y reutilización de los desarrollos informáticos por parte de la comunidad científica.

Concretando el objetivo principal de nuestro trabajo, se trata de mostrar cómo el UP, como procedimiento estandarizado, permite reducir el salto representacional entre el dominio del problema y el dominio de la solución (paso de la “realidad” al diseño y la implementación). Para ello, planteamos desde el estándar UP el desarrollo de un

framework en el contexto del *Modelo Lineal Generalizado* (MLG), utilizando sus artefactos (productos del proceso) para aportar argumentos que validen dicho objetivo.

Además, este *framework* ha de cumplir con unos requisitos generales que son los que creemos deben guiar cualquier nuevo sistema o subsistema estadístico informatizado. Así, en primer lugar, debe ser un sistema flexible y estable, que permita al profesional hacer uso de él para extender de manera fácil su funcionalidad. Por otro lado, debe estar preparado para el modelado estadístico y para la simulación estocástica, cumpliendo con requisitos de facilidad de uso y máxima eficiencia. A su vez, debe ser independiente de la interfaz de usuario, esto es, su diseño debe ser exportable a cualquier plataforma interactiva (interfaz de comandos, interfaz gráfica, etc.). Estas condiciones del *framework* se discuten con detalle en el apartado 6.2 de este trabajo.

En definitiva, dicho *framewok* debe ser extensible y de calidad, como características principales a destacar. En este sentido, es necesario disponer de herramientas de programación adscritas al paradigma de la OO, que permitan implementar diseños planificados bajo este enfoque.

Sin embargo, el simple uso de estas herramientas no garantiza que el material desarrollado a partir de ellas vaya a ser *extensible* o de *calidad*, tal como afirman Ocaña y Sánchez (2003). Por ello, sin entrar en consideraciones acerca de la conveniencia de uno u otro lenguaje de programación concreto, estos autores plantean una discusión en relación al uso de posibles *patrones de diseño* adecuados para el desarrollo de programas estadísticos que cumplan con estos criterios de calidad y de extensibilidad, elementos directamente relacionados con la orientación a objetos. Concretamente, esbozan la utilización de distintas estrategias (patrones) aplicables al diseño de software para la simulación estadística. Un *framework* que resuelva los problemas de diseño utilizando patrones es mucho más probable que consiga un grado mayor de reutilización del diseño y del código que otro que no se base en ellos.

De hecho, otro de los objetivos de este trabajo pasa por discutir que el diseño de un sistema es precisamente un factor crucial para conseguir que las características enunciadas en el párrafo anterior sean propiedades relevantes en el producto final, es decir, de la aplicación resultante de la implementación del diseño. En ese sentido, el diseño debe reflejar por sí

mismo dichas características, ser capaz de informar de las particularidades del sistema diseñado en cuanto a flexibilidad y capacidad de extensión. Así, la implementación de la aplicación pasa a un segundo plano de importancia, puesto que su diseño debe ser el factor que indique desde un primer momento si se han conseguido o no los objetivos de calidad planteados. Al final, el lenguaje de programación orientado a objetos elegido para la implementación no es la cuestión más relevante.

El marco del UP permite reducir riesgos en este sentido, asegurando que si se siguen las pautas marcadas por este estándar es muy probable que se alcancen estas características deseables en todo producto informático. El lenguaje UML es muy rico en detalles, aportando una sintaxis gráfica excelente para conseguir que el diseño de un determinado sistema "hable por sí mismo". Precisamente, pretendemos aprovechar la expresividad de UML para mostrar que el diseño de las soluciones que resuelven los requisitos del sistema es el núcleo de todo el proceso de desarrollo de software, puesto que permite corroborar, sin llegar a la implementación, si se están cubriendo de manera óptima los requisitos plasmados en el análisis del problema.

En síntesis, los objetivos de nuestro trabajo se concretan en:

1. Revisar en profundidad los procedimientos y herramientas necesarios para que un investigador en estadística adquiera la autonomía suficiente para diseñar (y opcionalmente implementar) extensiones de las plataformas ya existentes.

2. Analizar la adecuación del UP, como procedimiento estandarizado, para reducir el salto representacional entre el dominio del problema y el dominio de la solución, sea cual sea el contexto concreto de análisis de datos en que centre su interés.

3. Discutir el uso de "patrones de diseño" como factor crucial para conseguir que las características de calidad y extensibilidad sean propiedades relevantes en el producto final

Para la consecución de los objetivos marcados, la primera parte de este trabajo se centra en el marco teórico del modelado orientado a objetos en el UP, en la importancia del uso de patrones de diseño y en el uso de la notación UML como estándar de comunicación científica en este contexto.

A continuación en la parte empírica se plantea el modelado orientado a objetos, bajo las directrices teóricas enunciadas, del diseño de un *framework* que permita resolver demandas concretas en el contexto del modelado estadístico con MLG. La intención de esta parte empírica es discutir la adecuación de las pautas de desarrollo de software expuestas en el marco teórico para asegurar la calidad y extensibilidad de las aplicaciones estadísticas. Concretamente, en esta segunda sección se analizan los requisitos funcionales (de comportamiento) que debe asumir nuestro *framework* y se evalúan las diferentes aproximaciones o "soluciones" que desde la orientación a objetos se plantean en forma de patrones de diseño específicos –soluciones particulares a problemas recurrentes–. Cada uno de los patrones de diseño utilizados se ilustra con código Java, de forma coherente con el citado entorno de desarrollo estadístico Omegahat.

PARTE I. MARCO TEÓRICO

2. INTRODUCCIÓN AL MODELADO ORIENTADO A OBJETOS

2.1. OBJETIVOS

La intención principal de este apartado introductorio al concepto de modelado orientado a objetos es la de mostrar una panorámica inicial de todos aquellos aspectos que han permitido que el paradigma de la Orientación a Objetos (OO) sea actualmente una necesidad en la industria de desarrollo de software. La necesidad de rentabilizar al máximo todo el esfuerzo y dinero que las empresas invierten en la construcción de programas informáticos, requiere de un proceso de modelado de soluciones concretas ante problemas específicos. Ese proceso de modelado necesita además integrarse en el paradigma de la OO para conseguir la flexibilidad y eficiencia que demanda actualmente la evolución continua de los sistemas ya desarrollados. En otras palabras, no sería rentable generar un programa informático altamente eficaz y eficiente para solventar una necesidad determinada si cuando cambia en algún sentido esa necesidad hay que redefinir por completo el sistema para que vuelva a ser útil. En este sentido, la reutilización de componentes ya creados es una de las virtudes más destacables de la OO, pues permite realizar cambios mínimos en los sistemas para adaptarse a los nuevos requisitos. El objetivo principal de este apartado es, por tanto, convencer de las virtudes ligadas al proceso de modelado orientado a objetos en el desarrollo de software, frente al proceso de creación de software asociado al paradigma tradicional de la programación estructurada. Estas virtudes, de las que se habla en los siguientes apartados, permiten que el trabajo de unos sea aprovechado por otros en términos acumulativos.

2.2. INTRODUCCIÓN AL CONCEPTO DE MODELADO

Siguiendo a Booch et al. (1999), se puede decir que el modelado es una parte central de todas las actividades que conducen a la programación de buen software. Se construyen modelos para comunicar la estructura deseada y el comportamiento del sistema. Estos modelos permiten, además, visualizar y controlar la arquitectura del sistema. Construimos modelos también con la finalidad de comprender mejor el sistema que estamos

construyendo, muchas veces descubriendo oportunidades para la simplificación y la reutilización. En definitiva, se construyen modelos para controlar el riesgo.

Estos mismos autores (Booch et al., 1999) plantean un símil extraído del contexto de la construcción de inmuebles para convencer de la importancia de modelar durante el proceso de desarrollo de software. En este sentido, afirman que, curiosamente, una gran cantidad de empresas de desarrollo de software comienzan queriendo construir “rascacielos”, pero enfocan el problema como si estuvieran enfrentándose a la “caseta de un perro”. Si realmente se quiere construir el software equivalente a una casa o a un rascacielos, el problema es algo más que una cuestión de escribir grandes cantidades de software. De hecho, el software de calidad se centra en optimizar al máximo el código, imaginar como escribir menos software y conseguir el mejor rendimiento sin que disminuya la eficacia. Esto convierte al desarrollo de software de calidad en una cuestión de arquitectura, proceso y herramientas, es decir, en una cuestión de modelado.

Un modelo es una simplificación de la realidad, que se construye para comprender mejor el sistema que se está desarrollando. De hecho, se construyen modelos de sistemas complejos porque no es posible comprender el sistema en su totalidad.

A través del modelado, se consiguen cuatro objetivos:

1. Ayudar a visualizar cómo es o debería ser un sistema.
2. Especificar la estructura o el comportamiento de un sistema.
3. Proporcionar plantillas que guíen la construcción de un sistema.
4. Documentar las decisiones adoptadas.

Por otro lado, existen una serie de principios básicos de modelado (Booch et al., 1999). En primer lugar, es importante tener en cuenta que la elección de los modelos a crear tiene una profunda influencia sobre cómo se acomete un problema y cómo se da forma a una solución. En este sentido, los modelos adecuados pueden arrojar mucha luz sobre problemas de desarrollo muy complicados, ofreciendo una comprensión inalcanzable por otras vías; en cambio, los modelos erróneos desorientarán, haciendo que uno se centre en cuestiones irrelevantes. En el software, los modelos elegidos pueden afectar mucho a nuestra visión del mundo. Si se construye un sistema con la mirada de un analista que se basa en una perspectiva estructurada, probablemente se obtendrán modelos centrados en

los algoritmos, con los datos fluyendo de proceso en proceso. Si se construye, en cambio, con la mirada de un desarrollador orientado a objetos, se obtendrá un sistema cuya arquitectura se centra en una gran cantidad de clases y los patrones de interacción que gobiernan cómo trabajan juntas esas clases. Cada visión del mundo conduce a un tipo de sistema diferente, con diferentes costes y beneficios, aunque la experiencia sugiere que la visión orientada a objetos es superior al proporcionar arquitecturas flexibles.

Un segundo principio básico del modelado nos dice que todo modelo puede ser expresado a diferentes niveles de precisión. Un analista o un usuario final se centrarán en el qué; un desarrollador se centrará en el cómo. En cualquier caso, los mejores tipos de modelos son aquéllos que permiten elegir el grado de detalle, dependiendo de quién está viendo el sistema y por qué necesita verlo.

Un tercer principio establece que los mejores modelos están ligados a la realidad. En el desarrollo de software, el talón de Aquiles de las técnicas de análisis estructurado es la existencia de una desconexión básica entre el modelo de análisis y el modelo de diseño del sistema. No poder salvar este abismo hace que el sistema concebido y el sistema construido diverjan con el paso del tiempo. En los sistemas orientados a objetos, en cambio, es posible conectar todas las vistas casi independientes de un sistema en un todo semántico.

Finalmente, un cuarto principio establece que un único modelo no es suficiente, de manera que cualquier sistema se aborda mejor a través de un pequeño conjunto de modelos casi independientes. La expresión “casi independientes” en este contexto significa tener modelos que se puedan construir y estudiar separadamente, pero aún así relacionados. Por poner un ejemplo, cuando se está construyendo un edificio se necesitan distintos modelos de planos: planos de planta, alzados, planos de electricidad, planos de calefacción y planos de cañerías; por supuesto, se pueden estudiar los planos eléctricos de forma aislada, pero también podemos ver su correspondencia con los planos de planta, incluso su interacción con los recorridos de las tuberías en el plano de fontanería.

Este último principio es igualmente cierto para los sistemas de software orientados a objetos, de los que se habla en el siguiente apartado. Para comprender la arquitectura de tales sistemas, se necesitan vistas complementarias y entrelazadas: una vista de casos de

uso (que muestre los requisitos del sistema), una vista de diseño (que capture el vocabulario del espacio del problema y del espacio de la solución), una vista de procesos (que modele la distribución de los procesos e hilos [*threads*] del sistema), una vista de implementación (que se ocupe de la realización física del sistema) y una vista de despliegue (que se centre en cuestiones de ingeniería del sistema). Cada una de estas vistas puede tener aspectos tanto estructurales como de comportamiento. En conjunto, estas vistas representan los planos del software (figura 1).

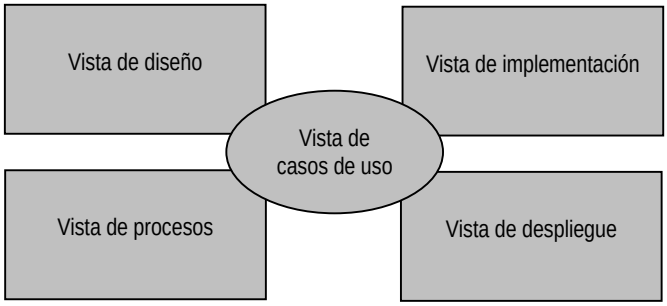


Figura 1. Modelado de la arquitectura de un sistema
(Fuente: Booch et al., 1999)

Según la naturaleza del sistema, algunos modelos pueden ser más importantes que otros. Por ejemplo, en sistemas con grandes cantidades de datos, dominarán los modelos centrados en las vistas de diseño estáticas. En sistemas con uso intensivo de interfaces gráficas de usuario (GUI), las vistas de casos de uso estáticas y dinámicas son bastante importantes. En los sistemas de tiempo real muy exigentes (por ejemplo, un sistema de control de tráfico), las vistas de procesos dinámicos tienden a ser más importantes. Finalmente, en los sistemas distribuidos, como los encontrados en aplicaciones de uso intensivo de la Web, los modelos de implementación y despliegue son los más importantes.

Llegados a este punto, queda justificada la necesidad de recurrir al modelado para obtener una representación inteligible de los requisitos y diseño del sistema que se desea implementar. En este sentido, se analizará en su momento cuáles de las posibles vistas representadas en la figura anterior (una vista es una proyección de un modelo) son más importantes en el proceso de desarrollo de software estadístico.

Además, en los siguientes apartados de este trabajo se justificará la idoneidad de plantear el desarrollo de software desde el paradigma orientado a objetos, destacando sus ventajas frente al desarrollo de software basado en el paradigma de programación estructurada. Por supuesto, esta idoneidad de la metodología orientada a objetos queda vinculada también al desarrollo de software estadístico.

La combinación del empleo de modelos que representan la realidad desde distintos puntos de vista (figura 1), asociada al uso de la metodología orientada a objetos, nos lleva a hablar de una serie de pasos centrales en el modelado orientado a objetos (apartado 3). Además, estos pasos aparecen reorganizados junto a otros en lo que se ha dado a conocer como el *Proceso Unificado de Desarrollo de Software* (Jacobson el al., 2000), proceso normalizado que describimos en el apartado 4. Por último, en el apartado 5, dentro de este marco teórico, se presenta un lenguaje específico de modelado también estandarizado y que se ha dado a conocer como el *Lenguaje Unificado de Modelado* (UML) (Booch et al., 1999). Este lenguaje permite representar gráficamente las diferentes vistas o modelos de un sistema orientado a objetos.

En el siguiente apartado, se conceptualiza el modelado orientado a objetos como una tecnología que se apoya en sólidos fundamentos de ingeniería del software (fundamentos descritos en el punto 2.3.4).

2.3. INGENIERÍA DEL SOFTWARE

La *informática* es la rama de la ciencia de la técnica que se ocupa de los ordenadores, tanto en la vertiente de arquitectura de los sistemas informáticos, como en la vertiente de diseño de técnicas para el tratamiento de la información.

En general, se distingue entre los componentes físicos de un ordenador, denominados *hardware*, y los componentes lógicos o programas que dirigen su funcionamiento, denominados *software*. Sin embargo, actualmente la frontera entre el software y el hardware se está difuminando, debido a la tendencia a almacenar en procesadores especializados el software de control de los dispositivos del ordenador. Esta tendencia se

verá acentuada con la aparición de los ordenadores de quinta generación (Moto-Oka y Kitsuregawa, 1986; Simons, 1985).

Para el desarrollo del software se necesitan técnicas capaces de crear productos muy complejos, que satisfagan además estándares estrictos de prestaciones y calidad, de acuerdo a una planificación, control y presupuesto adecuados. Los métodos de trabajo que se han establecido para responder a estas necesidades abarcan el ciclo de vida completo de un producto software (especificación, análisis, diseño, implementación, comprobaciones, uso y mantenimiento), y constituyen lo que se ha dado en llamar *ingeniería del software*.

2.3.1. Apuntes históricos

La evolución del software ha experimentado modificaciones importantes en su corta historia, que se pueden atribuir tanto al desarrollo de metodologías del propio software, como a la evolución del hardware. Desde la perspectiva actual, existe un aspecto que ha supuesto un punto de inflexión en la historia de estas tecnologías: «la crisis del software». Para comprender los elementos que han provocado esta crisis y las claves que han permitido superarla, es imprescindible hacer alusión a los avances que se han ido produciendo en esta área (Losilla, 1995).

En la década de los 50 y principios de los 60 se dispone de los primeros ordenadores de uso general, y las aplicaciones que se desarrollan están enfocadas al proceso por lotes (*batch*), con un mínimo de distribución y habitualmente diseñadas a medida de un cliente. En estos momentos, los métodos formales de construcción de software son prácticamente inexistentes, conceptuándose este proceso casi como un arte (Peralta y Rodríguez, 1994, cap. 1).

En la década de los 60 y principios de los 70 se empiezan a construir sistemas multiusuario y de tiempo real. También se desarrollan los primeros sistemas de gestión de bases de datos y nace el concepto de «producto de software». Paulatinamente, los sistemas se van ampliando y complicando, apareciendo problemas de mantenimiento del software, debido a que consume una gran cantidad de recursos humanos, hasta el punto de impedir en

algunas empresas la creación de nuevas aplicaciones. Es en estos momentos cuando se empieza a hablar de la «crisis del software» (Brooks, 1975).

En las décadas de los 70 y 80 se introduce el microprocesador, los ordenadores personales, las estaciones de trabajo, las redes de ordenadores y las arquitecturas cliente-servidor. Respecto al software, en esta etapa el nivel de creación y distribución de productos aumenta hasta el orden de cientos de miles o millones de copias, agudizándose la crisis, al tiempo que surgen los primeros auxilios por la vía de los sistemas de desarrollo distribuido y de las herramientas CASE (*Computer Aided/Assisted Software/System Engineering*), que son herramientas y metodologías que soportan un enfoque de ingeniería en todas las fases de desarrollo del software.

En la actualidad, el entorno informático se caracteriza por la consolidación de nuevos paradigmas de construcción de software, como la Orientación a Objetos (OO), la generalización de los Interfaces Gráficos de Usuario (IGU), la extensión de la Inteligencia Artificial (IA) y de los sistemas expertos, las redes neuronales, los algoritmos genéticos, etc. En cuanto a hardware, cabe destacar el procesamiento en paralelo, el aumento en el nivel de integración de los circuitos (VLSI) y la arquitectura de flujo de datos y de reducción de gráficos, que están permitiendo una capacidad de proceso inimaginable hace pocos años.

Entre estos elementos, cabe destacar la OO, que en el contexto de la ingeniería del software está aportando la claves para solucionar la crisis a la que nos venimos refiriendo. Hoy en día, la ingeniería del software se concibe como una extensión de la ingeniería de sistemas y de los ordenadores, que engloba tres elementos principales: métodos, herramientas y procedimientos. Los *métodos* permiten afrontar las tareas de planificación, especificación, análisis, diseño, implementación, pruebas y mantenimiento del sistema informático. Las *herramientas* suministran un soporte automatizado o semi-automatizado a los métodos. Por último, los *procedimientos* unen los métodos y las herramientas, y permiten obtener los objetivos propuestos en el tiempo y la calidad requeridos.

Tanto los métodos como las herramientas y los procedimientos, se enmarcan en un esquema conceptual general o paradigma que rige el modelado de la realidad que se persigue.

Para abordar y construir sistemas de primera generación, cuyo objetivo era la automatización de procesos claramente definidos y totalmente observables, el paradigma adecuado ha sido el denominado «ciclo de vida clásico». En cambio, en los sistemas de segunda y tercera generación se desea obtener un nivel de servicios del ordenador cualitativamente más elevado, que no suele admitir una especificación inicial exacta. Además, los proyectos se acostumbran a abordar dividiéndolos en proyectos más pequeños, que se distribuyen entre varios grupos de trabajo. El software orientado a objetos admite de forma natural esta organización del desarrollo y, además, permite que los objetos independientes se ejecuten simultáneamente en procesadores paralelos (Martín y Odell, 1994).

En síntesis, para superar la crisis ha sido imprescindible que el software soporte en su propio diseño la adaptación constante a los cambios que puede sufrir la realidad que modela, en lugar de ser construido a medida para cada caso. Para ello, se ha evolucionado desde el diseño modular y la programación estructurada (perspectiva algorítmica), hasta el enfoque más moderno basado en la OO.

2.3.2. Perspectiva algorítmica versus *perspectiva orientada a objetos*

Los problemas que se intentan resolver con el software conllevan a menudo elementos de complejidad ineludible, en los que se encuentra una cantidad muy grande de requisitos que compiten entre sí, que quizás incluso se contradicen.

Los usuarios suelen enfrentarse a grandes dificultades al intentar expresar con precisión sus necesidades en una forma que los desarrolladores puedan comprender. En casos extremos, los usuarios pueden no tener más que ideas vagas de lo que desean de un sistema de software. Esto no es en realidad achacable a los usuarios ni a los desarrolladores del sistema; más bien ocurre porque cada uno de los grupos no suele conocer suficientemente el dominio del otro. Los usuarios y los desarrolladores tienen perspectivas diferentes sobre la naturaleza del problema y realizan distintas suposiciones sobre la naturaleza de la solución. La forma habitual de expresar necesidades hoy en día es mediante grandes cantidades de texto, ocasionalmente acompañadas de unos pocos dibujos. Estos

documentos son difíciles de comprender, están abiertos a diversas interpretaciones, y demasiado frecuentemente contienen elementos que invaden el diseño en lugar de limitarse a ser requisitos esenciales.

Una complicación adicional es que los requisitos de un sistema de software cambian frecuentemente durante su desarrollo, especialmente porque la mera existencia de un proyecto de desarrollo de software altera las reglas del problema. La observación de productos de las primeras fases, como documentos de diseño y prototipos, y la posterior utilización de un sistema cuando ya está instalado y operativo, son factores que llevan a los usuarios a comprender y articular mejor sus necesidades reales. Al mismo tiempo, este proceso ayuda a los desarrolladores a comprender el dominio del problema, capacitándoles para responder mejor a preguntas que iluminan los rincones oscuros del comportamiento de un sistema.

Ya que un sistema grande de software es una inversión considerable, no es admisible el desechar un sistema existente cada vez que los requerimientos cambian. Esté o no previsto, los sistemas grandes tienden a evolucionar en el tiempo.

La tarea fundamental del equipo de desarrollo de software es dar vida a una ilusión de simplicidad –para defender a los usuarios de esta vasta y a menudo arbitraria complejidad externa–. Ciertamente, el tamaño no es una gran virtud para un sistema de software. Se hace lo posible por escribir menos código mediante la invención de mecanismos ingeniosos y potentes que dan esta ilusión de simplicidad, así como mediante la reutilización de marcos estructurales de diseños y código ya existentes. Sin embargo, a veces es imposible eludir el mero volumen de los requerimientos de un sistema y se plantea la obligación de o bien escribir una enorme cantidad de nuevo software o bien reutilizar software existente de nuevas formas. Hoy en día no es extraño encontrar sistemas ya terminados cuyo tamaño se mide en cientos de miles, o incluso millones de líneas de código (y todo esto en un lenguaje de programación de alto nivel, además).

La técnica de dominar la complejidad se conoce desde tiempos remotos: *divide et impera* (divide y vencerás). Cuando se diseña un sistema de software complejo, es esencial descomponerlo en partes más y más pequeñas, cada una de las cuales se puede refinar entonces de forma independiente. De este modo se satisface la restricción fundamental que

existe sobre la capacidad de canal de la comprensión humana: para entender un nivel dado de un sistema, basta con comprender unas pocas partes (no necesariamente todas) a la vez.

Descomposición algorítmica. La visión tradicional del desarrollo de software toma una perspectiva algorítmica. En este enfoque, el bloque principal de construcción de todo el software es el procedimiento o función. La programación en estadística y otras disciplinas científicas se ha basado normalmente en esta perspectiva estructurada o procedimental. Bajo este estilo de programación, la unidad base de ejecución son programas o conjuntos de instrucciones ejecutables, que se dividen en módulos o rutinas (utilizando terminología informática). Una aplicación suele estar formada por una jerarquía más o menos definida de programas y módulos que se pueden llamar unos a otros. Esta jerarquía se organiza en torno a un programa principal, desde el cual se accede a otros módulos, llamados subrutinas o subprogramas. Los datos tienen un papel secundario, y no son más que aquello con lo que se alimenta a los programas para que realicen su función. La programación estructurada se puede resumir en la siguiente expresión:

Algoritmos + Estructura de datos = Programas

En este paradigma de programación la clave reside en decidir qué procedimientos se quieren y en utilizar los mejores algoritmos que se encuentren. Los lenguajes de programación estructurada dan soporte a este paradigma con facilidades para pasar argumentos a las funciones empleadas y para retornar valores desde estas funciones. *Fortran* fue el lenguaje de procedimientos original; otros lenguajes como *Algol60*, *Algol68*, *C* y *Pascal* fueron invenciones posteriores en la misma tradición.

Casi todos los informáticos han sido adiestrados en el dogma del diseño estructurado descendente, y por eso suelen afrontar la descomposición como una simple cuestión de descomposición algorítmica (descomposición de algoritmos grandes en otros más pequeños), en la que cada módulo del sistema representa un paso importante de algún proceso global. No hay nada inherentemente malo en este punto de vista, salvo que tiende a producir sistemas frágiles, puesto que cuando los requisitos cambian y el sistema crece, los sistemas construidos con un enfoque algorítmico se vuelven muy difíciles de mantener.

Descomposición orientada a objetos. La visión actual del desarrollo de software toma una perspectiva orientada a objetos. En este enfoque, el principal bloque de construcción de

todos los sistemas software es el objeto o clase. Para explicarlo sencillamente, un objeto es una cosa, generalmente traída del vocabulario del espacio del problema o del espacio de la solución; una clase es una descripción de un conjunto de objetos similares. Todo objeto tiene identidad (puede nombrarse o distinguirse de otra manera de otros objetos), estado (generalmente hay algunos datos asociados a él), y comportamiento (se le pueden hacer cosas al objeto, y él a su vez puede hacer cosas a otros objetos). Aunque ambos diseños resuelven el mismo problema, lo hacen de formas bastante distintas. En esta segunda descomposición, se ve el mundo como un conjunto de agentes autónomos (objetos) que colaboran para llevar a cabo algún comportamiento de nivel superior; no existen algoritmos concebidos como elementos independientes, en lugar de eso, son operaciones asociadas a los objetos pertenecientes al sistema. Cada objeto contiene su propio comportamiento bien definido. Los objetos hacen cosas, y a los objetos se les pide que hagan lo que hacen enviándoles mensajes. Puesto que esta descomposición está basada en objetos y no en algoritmos, se le llama descomposición *orientada a objetos*.

En contraste con la aproximación anterior, que centra el punto de atención en las funciones o rutinas como base de ejecución, en esta nueva aproximación son los datos en lugar de las funciones los que forman la jerarquía básica. La programación orientada a objetos fue concebida por personas que veían en su entorno no acciones sino objetos que interactuaban unos con otros según su naturaleza. Las acciones aplicadas a los objetos dependen de éstos, lo que representa, en términos informáticos, que los programas y subprogramas pasan a un nivel secundario, dependiendo de los datos. En este sentido, una clase son unos datos y unos métodos que operan sobre esos datos.

En síntesis, la programación orientada a objetos puede ser resumida de la siguiente manera:

Objetos + Flujo de mensajes = Programas

En el apartado 2.3.3 se habla de la evolución desde sus orígenes del paradigma de la orientación a objetos y, en consecuencia, de los distintos lenguajes de programación orientados a objetos conocidos por su papel en dicha evolución.

Descomposición algorítmica versus descomposición orientada a objetos. ¿Cuál es la forma correcta de descomponer un sistema complejo, por algoritmos o por objetos? Según Booch (1996), la respuesta adecuada es que ambas visiones son importantes: la visión

algorítmica enfatiza el orden de los eventos, y la visión orientada a objetos resalta los agentes que o bien causan acciones o bien son sujetos de estas acciones. Sin embargo, el hecho es que no se puede construir un sistema complejo de las dos formas a la vez, porque son vistas completamente perpendiculares. Por otra parte, la descomposición orientada a objetos tiene una serie de ventajas altamente significativas sobre la descomposición algorítmica. La descomposición orientada a objetos produce sistemas más pequeños a través de la reutilización de mecanismos comunes, proporcionando así una importante economía de expresión. Los sistemas orientados a objetos son también más resistentes al cambio y por tanto están mejor preparados para evolucionar en el tiempo, porque su diseño está basado en formas intermedias estables. En realidad, la descomposición orientada a objetos reduce en gran medida el riesgo que representa construir sistemas de software complejos, porque están diseñados para evolucionar de forma incremental partiendo de sistemas más pequeños en los que ya se tiene confianza. Es más, la descomposición orientada a objetos resuelve directamente la complejidad innata del software ayudando a tomar decisiones respecto a la separación de intereses en un gran espacio de estados.

2.3.3. La Orientación a Objetos (OO) como paradigma en el desarrollo de software

La Orientación a Objetos (OO), por tanto, es una nueva forma de entender el desarrollo del software, que abarca un conjunto de metodologías y herramientas para el modelado y la implementación del software, las cuales hacen más fácil la construcción de sistemas complejos a partir de componentes individuales.

La génesis de las ideas básicas de la OO se produce a principios de los años 60 y se atribuye al trabajo del Dr. Nygaard y su equipo de la Universidad de Noruega. Estos investigadores se dedicaban al desarrollo de sistemas informáticos para simular sistemas físicos, tales como el funcionamiento de motores. La dificultad con la que se encontraban era doble: los programas eran muy complejos y forzosamente tenían que ser muy modificables. Este segundo punto era especialmente problemático, ya que eran muchas las ocasiones en las que se requería probar la viabilidad y el rendimiento de estructuras alternativas.

La solución que idearon fue diseñar el programa con una estructura paralela a la del sistema físico. Es decir, si el sistema físico estaba compuesto por cien componentes, el programa también tenía cien módulos, uno por cada pieza. Partiendo el programa de esta manera, había una total correspondencia entre el sistema físico y el sistema informático, dado que cada pieza tenía implementada su *abstracción* en un módulo informático y que los módulos se comunicaban enviándose *mensajes*, de la misma forma que los sistemas físicos se comunican enviándose señales. Para dar soporte a estas ideas crearon un lenguaje denominado Simula-67 (Dahl, Myrhaug y Nygaard, 1968; Dahl y Nygaard, 1966).

Este enfoque resolvió los dos problemas planteados. En primer lugar, ofrecía una forma natural de dividir un programa muy complejo en partes más sencillas y, en segundo lugar, se simplificaba considerablemente el mantenimiento de dicho programa, permitiendo al investigador el cambio de piezas enteras, o la modificación del comportamiento de alguna de ellas, sin tener que alterar el resto del programa.

En la década de los años 70, Alan Kay, de la Universidad de Utah, formó un grupo de investigación junto con Adele Goldberg y Dan Ingalls de Xerox (Palo Alto). Este grupo diseñó un entorno y un lenguaje de programación llamado *Smalltalk*, que incorporaba las ideas de la orientación a objetos (Savic, 1990). En la década de los 80, Bjarne Stroustrup, de ATT-Bell, parte de *Smalltalk* y *Simula* para diseñar el lenguaje *C++*, como sucesor del *C* (Stroustrup, 1986). En el ámbito académico, Bertrand Meyer (1991) crea el lenguaje *Eiffel*, reconocido como el más completo y elegante de los lenguajes orientados a objetos (LOO). Paralelamente, en el mundo de la inteligencia artificial se desarrolla, entre otros, el lenguaje *CLOS* (*Common Lisp Object System*), como variante orientada a objetos del lenguaje Lisp (Moon, 1989; Steel, 1990).

Actualmente, sin duda, uno de los lenguajes con mayor impacto en la industria de desarrollo de software es el lenguaje *Java*, lenguaje de programación OO desarrollado por Sun Microsystems en 1995. Sun Microsystems lanzó el entorno JDK 1.0 en 1996 (Gosling, Joy y Steele, 1996), primera versión del kit de desarrollo de dominio público que se convirtió en la primera especificación formal de la plataforma Java. Desde entonces han aparecido diferentes versiones, aunque la primera comercial se denominó JDK 1.1 y se lanzó a principios de 1997. En diciembre de 1998 Sun lanzó la plataforma Java 2 (conocida

también como JDK 1.2). Esta versión ya representó la madurez de la plataforma Java, aunque se han seguido incorporando hasta la fecha nuevas funcionalidades; en este sentido, es posible acceder a Java 2 JDK 1.4 desde febrero de 2002.

La popularidad y aceptación general de las tecnologías orientadas a objetos comenzó a producirse en la década de los 90. Aparecieron nuevas revistas enteramente dedicadas a la OO, como el *Journal of Object-Oriented Programming*, el *Object Magazine*, etc., y se comenzaban a celebrar congresos y conferencias especializadas, como la *OOPSLA*, centrada en los sistemas de programación y lenguajes orientados a objetos. También han ido surgiendo diversos esfuerzos de estandarización en el ámbito de las metodologías orientadas a objetos, el más notable de los cuales es, sin duda, el OMG (*Object Management Group*), refrendado por la mayoría de las empresas e instituciones desarrolladoras de software. OMG produce y mantiene una serie de especificaciones, las cuales sustentan proyectos de desarrollo de software desde el análisis y diseño hasta la implementación, instalación, comprobaciones en tiempo de ejecución y mantenimiento.

En este sentido, dentro del marco de la metodología OMG cobra gran importancia la especificación UML (*Unified Modeling Language*), que estandariza la representación del análisis y diseño orientado a objetos. UML es un lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software, así como para el modelado de negocios (Booch et al., 1999). Esta especificación proporciona una forma estándar de describir los planos de un sistema, cubriendo tanto los aspectos conceptuales, tales como procesos del negocio y funciones del sistema, como los aspectos concretos, tales como las clases escritas en un lenguaje de programación específico, esquemas de bases de datos y componentes software reutilizables. UML nació en 1994 como una fusión de los métodos y las notaciones de G. Booch y J. Rumbaugh, a los que posteriormente se unió I. Jacobson. UML 1.0 se ofreció para su estandarización al *Object Management Grup* (*OMG*) en enero de 1997, en respuesta a su solicitud de propuestas para un lenguaje estándar de modelado. OMG ha asumido desde entonces el control de mantenimiento de UML, especificación que actualmente se encuentra en su versión 1.5 (OMG, 2003).

Una vez revisada la génesis de la OO y su estado actual, volvemos a conectar con las ideas básicas de este paradigma de programación orientado a objetos. En síntesis, dicho paradigma pone el énfasis en el aspecto de *modelado* del sistema, examinando el domino del problema como un conjunto de objetos que se comunican entre sí mediante mensajes. De este modo, la estructura de los programas refleja directamente la del problema que se desea simular.

Utilizando las metodologías de análisis y diseño orientado a objetos (AOO y DOO, respectivamente) y sus lenguajes (LOO), el software se construye a partir de *objetos* con un comportamiento específico. Los propios objetos se pueden construir a partir de otros, que a su vez pueden estar formados por otros objetos. Este proceso conlleva un beneficio muy importante, y que constituye la razón fundamental por la cual la ingeniería del software se ha abocado a este nuevo paradigma: la *reusabilidad*.

Cuando se construye un nuevo programa se obtienen piezas para futuros programas, lo cual conduce a que el software se elabore por ensamblamiento de objetos desarrollados por uno mismo o por otros para otras aplicaciones. Estos objetos pueden ser cada vez más complejos desde el punto de vista interno, pero más sencillos en cuanto a la interacción con ellos, ya que se pueden conceptuar como «cajas negras» susceptibles de ser utilizadas sin mirar en su interior. Lo que se persigue es disponer de depósitos con una gran cantidad de objetos, y de herramientas que permitan encontrar el tipo necesario de objeto, reutilizarlo y ampliar su comportamiento de la forma más conveniente.

Por último, cabe destacar que las características del desarrollo de software orientado a objetos permiten reducir la posibilidad de incurrir en errores durante la programación o la reprogramación. Hay que recordar que, no en vano, uno de los adagios más populares en el entorno de la ingeniería del software es el que afirma que *"el software no se escribe, se reescribe"*, y que algunos de los errores más importantes que se han descrito a lo largo de la historia de la informática se relacionan con programas que han fallado tras realizar costosas modificaciones en ellos.

2.3.4. Principios de la Orientación a Objetos

En este apartado se exponen, de forma sintética, los tres conceptos o principios básicos de la OO:

- Encapsulado y ocultación de la información.
- Clasificación, tipos abstractos de datos y herencia.
- Polimorfismo.

a. Encapsulado y ocultación de la información

En las metodologías tradicionales orientadas al proceso, como el análisis y el diseño estructurado de De Marco (1982), se produce una dicotomía entre los dos elementos constituyentes de un sistema: *funciones* que llevan a cabo los programas y *datos* que se almacenan en ficheros o bases de datos. En la OO, sin embargo, se propugna un enfoque unificador de ambos aspectos, que se *encapsulan* en los objetos, con la finalidad de modelizar y representar mejor el mundo real (Booch, 1986, 1991, 1996; Stroustrup, 1988).

A nivel conceptual, un *objeto* es una entidad percibida en el sistema que se está desarrollando, mientras que a nivel de implementación, un objeto se corresponde con el encapsulado de un conjunto de operaciones (habitualmente denominadas métodos o servicios) que pueden ser invocados externamente, y de un conjunto de variables (habitualmente denominadas atributos) que almacenan el estado resultante de dichas operaciones. Sólo el propio objeto tiene la capacidad de acceder y modificar sus datos, mediante los métodos que tiene implementados. En orientación a objetos, se habla de *evento* cuando un método modifica un atributo de estado.

El encapsulado permite *ocultar* a los usuarios de un objeto los aspectos instrumentales (la propia programación), ofreciéndoles únicamente una interfaz externa mediante la cual interactuar con el objeto. Este principio de ocultación es fundamental, puesto que permite modificar los aspectos privados de un objeto sin que se vean afectados los demás objetos

que interactúan con él, siempre que se conserve la misma interfaz externa. Dicho de otro modo, el encapsulado proporciona al programador libertad en la implementación de los detalles de un sistema, con la única restricción de mantener la interfaz abstracta que ven los usuarios externos.

Martín y Odell (1994, p. 29) reproducen una interesante analogía propuesta por David Taylor (1992) entre los conceptos de la OO enunciadas hasta ahora y los manejados por la citología.

> *«David Taylor ha señalado que el diseño orientado a objetos refleja las técnicas de la naturaleza. Todos los seres vivos están compuestos por células. Las células son paquetes organizados, que al igual que los objetos, combinan la información y el comportamiento. La* ***información*** *de las células está en el ADN y en las moléculas de proteína del núcleo. Los* ***métodos*** *de la célula los realizan orgánulos que rodean al núcleo.*
> *La célula está cubierta por una membrana que protege y oculta la labor celular de cualquier intrusión del exterior. Las células no pueden "leer" las moléculas de proteína de las demás o controlar la estructura de las demás; solo "leen" y controlan lo propio. En vez de esto, envían solicitudes químicas a las demás células. Al empaquetar de esta manera la información y el comportamiento la célula se encapsula.*
> *Taylor comenta: "Esta comunicación basada en mensajes hace mucho más sencillo el funcionamiento celular... La membrana oculta la complejidad de la célula y presenta una interfaz relativamente sencilla al resto del organismo... Como se puede ver por la estructura celular, el* ***encapsulado*** *es una idea que ha estado latente durante mucho tiempo".»*

b. Clasificación, tipos abstractos de datos y herencia

La potente disciplina que subyace bajo el paradigma de la OO es la tipificación o clasificación de datos abstractos. Los tipos de datos abstractos, habitualmente denominados *clases*, definen conjuntos encapsulados de objetos reales conceptualmente similares. En los lenguajes orientados a objetos (LOO), las clases se utilizan para describir los tipos de datos

abstractos, y se reserva el término *objeto* para referir las realizaciones o casos concretos de las clases (*instances*, en inglés), que se generan durante la ejecución de los programas. Una clase define los datos que se están almacenando (variables o atributos) y las operaciones (métodos) soportadas por los objetos que son instancias de la clase. La idea principal de la OO es que un sistema orientado a objetos es un conjunto de objetos que interactúan entre sí y que están organizados en clases.

El concepto de clase (tipo abstracto de datos) nos lleva a la noción de *abstracción*, entendida como el proceso de capturar los detalles fundamentales de un objeto mientras se suprimen o ignoran los detalles. Booch (1996, p. 46) reproduce una afirmación de Hoare, que sugiere que «la abstracción surge de un reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y la decisión de concentrarse en esas similitudes e ignorar por el momento las diferencias» (Dahl, Kijkstra y Hoare, 1972). La abstracción proporciona un mecanismo crucial para permitir que las personas comprendan, se comuniquen y razonen sistemas complejos. Sin abstracción, el nivel de detalle requerido para comprender un sistema hace que las personas sean incapaces de construir sus modelos mentales de cómo se estructura el sistema y cómo funciona. La noción de abstracción entraña la descomposición de un sistema complejo o complicado en sus partes más fundamentales y la descripción de esas partes con un lenguaje sencillo y preciso. Por ejemplo, las personas no piensan en un coche como centenares de elementos, sino como un objeto bien definido con un comportamiento propio. Esta abstracción permite a las personas utilizar un coche para conducirlo sin tener que preocuparse de la complejidad de las partes que forman el coche; pueden ignorar los detalles de cómo funciona el motor, los frenos o el sistema de refrigeración. Realmente, al igual que sucede en el mundo real, lo que interesa es considerar el objeto como un todo. Al adoptar una visión del entorno orientada a objetos, el empleo de la abstracción permite utilizan bloques de información de contenido semántico cada vez mayor; esto es así puesto que los objetos, como abstracciones de entidades del mundo real, representan un agrupamiento de información particularmente denso y cohesivo (Booch, 1996).

Otra noción importante en los lenguajes orientados a objetos (LOO) es el concepto de *herencia*. Es posible derivar nuevas clases a partir de una clase dada o realizar (instanciar) directamente objetos mediante un proceso de *herencia*, concepto que se define de forma

parecida a la herencia en el sentido biológico. Se puede crear una nueva clase u objeto heredando los atributos y servicios de una o varias clases padre (herencia simple y múltiple, respectivamente). Las nuevas clases que se van creando por herencia configuran las denominadas «jerarquías de clases». En este contexto se utiliza el término «superclase» para referir cualquiera de las clases de orden superior en una misma jerarquía.

En el siguiente párrafo se ilustran los conceptos de *clase* y *herencia*, retomando la analogía propuesta por David Taylor (1992) entre la OO y el funcionamiento y estructura celular:

> *«Las células son un admirable bloque universal de construcción de la naturaleza. Existen células sanguíneas que transportan sustancias químicas, células del cerebro, células óseas, células que permiten el funcionamiento de la retina del ojo y células musculares que distorsionan su forma para llevar a cabo funciones mecánicas. Los componentes de todas las plantas, insectos, peces y mamíferos están constituidos de células de estructura común que actúan según ciertos principios básicos. En principio, todo el software se podría construir de manera análoga con ciertas* ***clases****. Aunque existe una gran diversidad de células, muchas de ellas, como los* ***objetos****, tienen tipos similares. Un tipo de célula puede operar de manera similar a otro, puesto que ambas han* ***heredado*** *propiedades semejantes con la evolución. Las células se agrupan en órganos, como los músculos o las uñas de los pies. Los órganos se agrupan en sistemas y aparatos como, por ejemplo, el sistema nervioso.*
>
> *Un organismo está compuesto por varios sistemas y aparatos. Aunque forme parte de un organismo complejo, cada célula actúa por su cuenta, como un objeto, sin conocer la razón por la que se le envía un mensaje o las últimas consecuencias de su comportamiento.»* (Martín y Odell, 1994, p. 29).

La herencia es un principio muy importante de la OO porque es precisamente en ella donde radica la *reusabilidad*, la *extensibilidad* y el *bajo coste* de mantenimiento de los sistemas informáticos basados en objetos. Generalmente, los LOO incluyen una vasta jerarquía

inicial de clases, a partir de la cual se pueden empezar a construir las aplicaciones. Cabe señalar que los LOO que no admiten la herencia múltiple, generalmente permiten simularla, por ejemplo, mediante *agregación* de clases.

Para facilitar la definición de las clases y objetos, y de las relaciones que se establecen entre ellas (ya sean de herencia o de agregación), se acostumbran a utilizar las tarjetas CRC (*Class, Responsability and Collaboration*) (Wirfs-Brock, Wilkerson y Wiener, 1990). La figura 2 presenta el diseño de tarjeta propuesto por Graham (1994a,1994b).

<table>
<tr><td colspan="2">Nombre de la clase:
abstracta / concreta
dominio / aplicación</td></tr>
<tr><td colspan="2">Superclases:</td></tr>
<tr><td colspan="2">Atributos y relaciones:</td></tr>
<tr><td>Servicios:</td><td>Servidores:</td></tr>
<tr><td colspan="2">Reglas:</td></tr>
</table>

Figura 2. Estructura de una tarjeta CRC (notación SOMA; Graham, 1994b)

Como se puede observar en la figura 2, las tarjetas CRC contienen toda la información relativa a las «responsabilidades» (atributos y servicios) de la clase, así como las «colaboraciones» o relaciones que ésta mantiene con otras clases. En la propuesta de Graham (1994b) se distingue si la clase es del «dominio» o de la «aplicación». Una clase del dominio implica que es persistente (está almacenada en disco) y que, probablemente, será estable en la estructura durante todo el tiempo de vida del programa. Las clases u objetos de la aplicación son más volátiles, creándose y destruyéndose en tiempo de ejecución. Las clases también pueden ser «abstractas» o «concretas», según permitan la derivación por herencia sólo de nuevas clases o también de objetos, respectivamente. Para

cada clase se indica, en el apartado «superclases», las clases de las cuales ha heredado parte de sus atributos y servicios.

También se especifica el nombre, visibilidad (público o privado), y tipo y valor por defecto (si existe) de cada uno de los «atributos» de la clase. Un atributo puede ser a su vez una clase, en cuyo caso se debe escribir el nombre de ésta, estableciéndose una relación de *agregación* entre ambas clases. La agregación denota una jerarquía todo/parte, con la capacidad de ir desde el todo (también llamado el *agregado*) hasta sus partes (conocidas también como *atributos*). A través de esta relación, el agregado puede enviar mensajes a sus partes.

Cada «servicio» se detalla con un nombre y una descripción de su algoritmo en pseudo-código. Cuando un servicio necesita acceder a otra clase u objeto para obtener información, se indica el nombre de dicha clase en el apartado «servidores». Además, se debe establecer si el servicio es público (se puede invocar desde clases de otras jerarquías), o pertenece al comportamiento privado de la clase (sólo puede ser invocado por clases u objetos que derivan de ella).

Volviendo al concepto de *encapsulado y ocultación de la información*, dado que el propósito de una clase es encapsular complejidad, hay mecanismos para ocultar la complejidad de la implementación dentro de la misma. Cada método (servicio) o variable (atributo) de una clase puede ser público o privado, como ya se ha indicado; en este sentido, la interfaz *pública* de una clase representa todo lo que los usuarios externos de la clase necesitan conocer o pueden conocer. Los métodos y variables *privados* sólo pueden accederse por el código que es miembro de la clase. Por consiguiente, cualquier código que no es miembro de la clase no puede acceder a un método o atributo privado. Booch (1996) expone que la abstracción y el encapsulado son conceptos que se complementan: la abstracción se centra en el comportamiento observable de un objeto, mientras que el encapsulado se centra en la implementación que da lugar a ese comportamiento, ocultando todos los secretos de un objeto que no contribuyen a sus características esenciales. El encapsulado proporciona barreras explícitas entre abstracciones diferentes y por tanto conduce a una clara separación de intereses.

Finalmente, retomando la descripción de la figura 2, las «reglas» que aparecen en la parte inferior de la tarjeta se indican con notaciones simples del tipo SI...ENTONCES y expresan, por ejemplo, las condiciones que se deben cumplir para ejecutar un servicio, o también, para resolver los conflictos y ambigüedades que pueden aparecer durante la ejecución del sistema debidos a la herencia múltiple. Estas «reglas» constituyen una parte fundamental de la definición de una clase, ya que, idealmente, el soporte completo de las clases requiere que sus operaciones sean completas y correctas. Puesto que la semántica completa de las clases sólo existe en la mente de quien la crea, en la práctica, la completitud o exactitud de la clase será tan buena como lo sea la completitud o exactitud del código que captura su conducta. En este sentido, para ayudar a expresar mejor la conducta de las clases, los LOO deben proporcionar construcciones que indiquen las *restricciones* que prueban la exactitud o completitud de las clases. Estas restricciones se pueden implementar mediante reglas asociadas a los atributos, como los «predicados anexionados» que se utilizan en algunos sistemas de inteligencia artificial (IA), o las «reglas de integridad» de los sistemas de bases de datos. Generalmente, estas reglas se activan automáticamente al acceder o modificar un atributo, con el fin de comprobar que los valores de dicho atributo no infringen alguna condición relevante para el sistema.

Existen herramientas CASE que permiten definir las clases y objetos, indicando sus características mediante plantillas a modo de tarjetas CRC, así como la notación gráfica que se desea utilizar. Con estas herramientas, cada elemento de la plantilla principal (atributo, servicio, regla, etc.) tiene asociada una plantilla específica en la que se detalla dicho elemento. Además, con las herramientas CASE se establecen gráficamente las relaciones de herencia, de agregación o de cualquier otro tipo que existan entre las clases. Asimismo, muchas de estas herramientas generan automáticamente código de programación en varios LOO. Entre las herramientas CASE que admiten esta funcionalidad se puede destacar *Rational Rose*, *OOTher*, *SOMA*, *WithClass*, etc.

c. Polimorfismo

Es posible definir una clase estableciendo de forma virtual algunos de sus atributos o servicios, a la espera de que se implementen en el momento de la realización de un objeto

concreto o de la derivación de nuevas clases. En este caso se puede aprovechar otra de las características más relevantes de la OO: el *polimorfismo*.

Así pues, un objeto que deriva de una clase que tiene definidos servicios virtuales, haciendo uso del polimorfismo puede enviar mensajes, con el tipo de servicio que se desea obtener, a otros objetos que sabe que se lo pueden proporcionar. El modo concreto en que se ejecutará el servicio depende únicamente del objeto receptor porque es el que incorpora el comportamiento. En la práctica, el polimorfismo es la propiedad que permite enviar el mismo mensaje a objetos de diferentes clases, de forma que cada uno de ellos responde a ese mismo mensaje de modo distinto dependiendo de su implementación; el polimorfismo se aplica sólo a métodos que tienen la misma *signatura* (nombre, tipo y número de argumentos) pero están definidos en clases diferentes. Conviene distinguir en este sentido el polimorfismo de lo que se conoce como *sobrecarga de métodos*, que ocurre cuando una clase tiene múltiples métodos con el mismo nombre, cada uno de ellos con una signatura distinta.

El polimorfismo es una de las características que permiten la *independencia de los datos*, otra de las grandes ventajas de la OO. Gracias al polimorfismo se pueden incorporar nuevas subclases en las aplicaciones, a partir de las cuales se podrán realizar nuevos tipos de objetos y aumentar la funcionalidad del sistema, sin requerir ninguna modificación del programa. Esta característica de los lenguajes orientados a objetos permite a los programadores separar los elementos que cambian de los que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.

El polimorfismo adquiere una relevancia especial en la práctica, ya que permite crear entornos de trabajo muy ricos en clases genéricas, específicos para determinados tipos de aplicaciones, denominados «*application frameworks*». Actualmente, uno de los principales objetivos de los ingenieros de software es el diseño de *frameworks* que permitan generar aplicaciones concretas a muy bajo coste, y que posibiliten la acumulación real del trabajo y de los conocimientos en un determinado campo de investigación o de aplicación. Así, por ejemplo, Anderson (1995), ha descrito un *framework* denominado *StatTools*, que incluye una serie de jerarquías de clases que se pueden utilizar para la implementación de simulaciones en el ámbito de la estadística. Otro ejemplo, esta vez en el ámbito de la

simulación de redes neuronales desde la perspectiva conexionista, es el *framework* que presenta Blum (1992), que contiene diversos algoritmos de aprendizaje modelados e implementados bajo el paradigma de la orientación a objetos.

2.3.5. Diseño orientado a objetos: ejemplo ilustrativo

Estos principios básicos de la OO se pueden concretar a través un ejemplo ilustrativo. En la figura 3 se muestra un ejemplo de diseño orientado a objetos bajo notación UML, obtenido con una herramienta CASE. En este diseño se expone un posible planteamiento de resolución de un problema de software relacionado con la construcción de figuras geométricas.

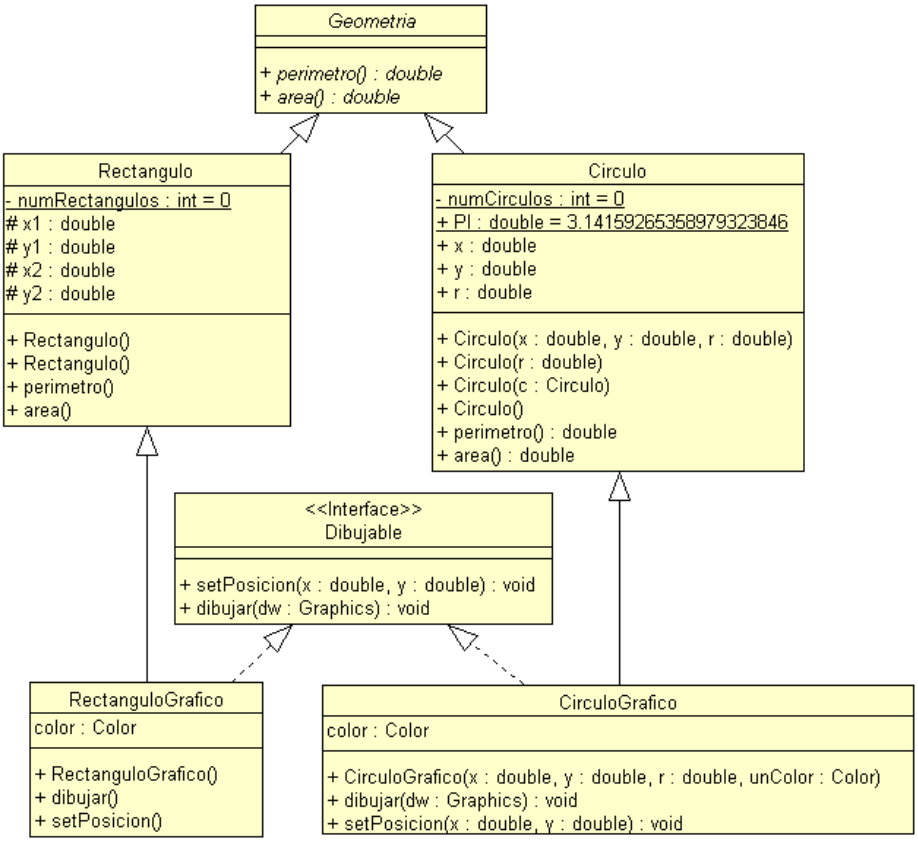


Figura 3. Diseño orientado a objetos (notación UML)[1]

[1] Se han ocultado los parámetros (entre paréntesis) de las clases *RectanguloGrafico* y *Rectangulo* por economía de espacio.

Con el análisis de un ejemplo concreto de diseño OO, se pretende mostrar una panorámica inicial de las ventajas asociadas al empleo de esta metodología en el planteamiento de soluciones adecuadas a demandas planteadas en el contexto del desarrollo de software. Además, el uso de la notación UML para representar la propuesta de solución de un problema, permite normalizar las pautas de comunicación entre diseñadores.

En este ejemplo (figura 3), observamos una serie de elementos a distinguir: clases, interfaces y flechas de relación. Las clases se representan en forma de rectángulo con tres compartimentos, el primero incluye el nombre de la clase, el segundo informa de los atributos de la clase (si procede) y el tercero incluye las operaciones (métodos) que se asociarán a los objetos que derivarán de las clases instanciables (clases no abstractas). Una interfaz (*interface* en inglés) es una colección de operaciones con un nombre, que se usa para especificar un servicio de clase. Se puede representar como una clase estereotipada (interfaz *Dibujable* en el ejemplo), pero, al contrario que las clases, las interfaces no especifican ninguna estructura (no pueden incluir atributos) ni especifican implementación (declaran sus operaciones, pero no se desarrollan en el momento de la codificación). Una *clase abstracta* (clase *Geometria*) es una clase de la que no se pueden crear objetos –su nombre se representa en cursiva en notación UML–. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas son similares a las interfaces, puesto que ninguna de las dos puede tener instancias directas de objetos y ambas proporcionan una serie de operaciones; sin embargo, una clase abstracta sí puede tener atributos, mientras que una interfaz no posee esta propiedad.

Queda claro, por tanto, que una interfaz no es una clase, y que especifica un comportamiento (operaciones o métodos) que va a tener la clase que la implemente. En el ejemplo, las clases *RectanguloGrafico* y *CirculoGrafico* implementan a la interfaz *Dibujable*, relación que queda plasmada a nivel gráfico mediante un tipo de flecha específica (punta de flecha cerrada y vacía, con línea de conexión discontinua y siempre en la dirección clase-interfaz). Esta relación implica, en definitiva, que la interfaz especifica un contrato para la clase sin dictar su implementación. Son todas las clases que implementan dicha interfaz las que están obligadas a proporcionar una definición de sus métodos en la etapa de implementación del diseño (etapa posterior a la de diseño).

Podemos observar efectivamente que las operaciones *setPosicion(x: double, y: double)* y *dibujar(dw: Graphics)* de la interfaz son recogidas por las clases que la implementan. Estas operaciones permitirán dibujar la figura geométrica en cuestión y establecer su posición en un contenedor de objetos que podríamos llamar *PanelDibujo* (cuya clase no incluimos en el diagrama por cuestiones de simplicidad). La operación *setPosicion( )* solicita los argumentos *x* e *y* (coordenadas), de tipo *double* (tipo de dato básico o primitivo), mientras que la operación *dibujar( )* incluye como argumento un objeto *dw*, de tipo *Graphics* (es decir, *dw* es una referencia a un objeto de la clase *Graphics*). Supuesto que la clase *Graphics* contiene una operación para establecer un color definido de línea y dispone también de una operación para dibujar un objeto gráfico con el color especificado, el objeto *dw* podrá emplear dichas operaciones para dibujar la figura en cuestión.

La construcción de un objeto dado se efectúa con una operación llamada *constructor*, operación especial que tiene el mismo nombre que el de la clase que generará dicho objeto durante la ejecución del programa. Por ejemplo, para dibujar un objeto *CirculoGrafico* con el método *dibujar( )*, en un objeto *PanelDibujo* ya creado, es necesario previamente construir dicho objeto con la operación *CirculoGrafico( )*, que es el constructor de la clase con el mismo nombre. Por supuesto, dicha operación puede solicitar al usuario determinados parámetros de construcción (parámetros delimitados entre paréntesis). Estos parámetros, para el caso de un objeto *CirculoGrafico* podrían ser los valores *x* e *y* que indiquen las dos coordenadas del centro del objeto, *r* para establecer su radio y el parámetro *unColor* para indicar el color de dicho objeto. El parámetro *unColor*, tal como vemos en la figura 3, representa una referencia a un objeto de la clase *Color*. Los valores de estos parámetros son empleados en la implementación del método *dibujar(dw: Graphics)* de la clase *CirculoGrafico*, utilizando para ello los métodos de la clase *Graphics*, métodos que generarán la figura con las propiedades indicadas por el usuario. Cabe anotar, en este sentido, que el atributo *color* de la clase *Color* es el que realmente se emplea en la implementación del método *dibujar( )*, cuyo valor (tipo de color) se le asigna en el constructor *CirculoGrafico( )* del mismo objeto, a través del parámetro *unColor* y utilizando para ello la expresión *color=unColor*.

Por otra parte, la implementación del proceso de construcción del objeto *CirculoGrafico*, de hecho, se lleva a cabo en este caso aprovechando los constructores de la clase *Circulo*.

Más en detalle, la clase derivada *CirculoGrafico* emplea su constructor *CirculoGrafico( )* para llamar a su vez a un constructor de la clase *Circulo*, que genera un objeto *Circulo* que contiene los parámetros cuyos valores se solicitarán posteriormente en la operación *dibujar( )*. Para poder implementar esta acción se ha establecido una relación de *herencia* entre la clase *CirculoGrafico* (la clase derivada) y la clase *Circulo* (la clase padre), representada a nivel gráfico con una flecha con punta cerrada y vacía, con línea de conexión continua y siempre en la dirección clase derivada-clase padre. En este ejemplo, aprovechamos esta herencia para llamar al constructor de la clase padre –supuesto que el lenguaje utilizado en la codificación permite esta opción–, obviando la implementación de dicho constructor en la clase derivada.

Se ha dicho explícitamente que la clase derivada (*CirculoGrafico*) llama en este caso a "un constructor" de su clase padre, precisamente porque la clase *Circulo* declara e implementa más de un constructor (mismo nombre que el de la clase). La única diferencia entre estos constructores reside en el número de parámetros que recogen. Como vemos, el constructor *Circulo(x:double, y:double, r:double)* permite manejar tres parámetros de tipo *double*, *x* e *y* para establecer las coordenadas del centro del objeto y *r* para recoger el valor de su radio. Lógicamente, para que el programa emplee este constructor se debe aportar el valor de cada uno de estos parámetros. El valor de los mismos se asignaría a los atributos *x, y, r*, respectivamente, definidos en el segundo compartimento de la clase *Circulo* (con propiedades de tipo *double*). El tipo *double* es un tipo de dato básico que especifica que la variable puede tomar valores reales (de coma flotante) y con precisión doble (ocupa ocho bytes en memoria). El tipo de dato asociado a un atributo o a un argumento especifica, por tanto, el rango de valores que puede tomar ese elemento.

Siguiendo con el resto de constructores de esta clase, si el usuario únicamente aporta el valor del parámetro *r*, el constructor empleado sería la operación *Circulo(r:double)*, y en su implementación se podría asignar a los atributos *x* e *y* el valor 0, por ejemplo, decisión ésta tomada lógicamente por el programador. Otro de los constructores emplea en la construcción de un objeto *Circulo* otro objeto de la misma clase (*Circulo(Circulo c)*), del cual saca una copia. Finalmente, si no se aporta ningún argumento (constructor *Circulo( )*), el programador puede asignar en este caso unos valores por defecto (por ejemplo valor 0 para *x* e *y*, y valor 1 para *r*). Este último es el llamado *constructor por defecto*. El empleo

de varios constructores en una clase es un ejemplo típico de *métodos sobrecargados*. Con este tipo de métodos, el compilador del programa puede determinar antes de su ejecución cuál es el método a llamar basado en el número y tipos de datos de los parámetros del método.

Retomando el tema de la herencia, se puede decir que la ventaja principal de heredar atributos y operaciones de una clase padre no es la de emplear dichos atributos y operaciones en la clase derivada, puesto que esto se puede hacer en cualquier clase no derivada (siempre y cuando estos atributos y propiedades hayan sido declarados públicos). La ventaja principal para la clase derivada es la de poder *redefinir* cualquiera de estos atributos y operaciones heredados, mientras se aprovecha el resto de funcionalidad no redefinida.

La acción de redefinir un método concreto que ha sido heredado, supone aportar nueva funcionalidad a este método (mismo nombre, con el mismo número y tipo de argumentos), pero en una nueva clase (derivada). De la misma forma, a partir de esta misma clase padre se podría derivar otra nueva clase, en la cual volver a redefinir el mismo método. Como consecuencia, podemos hacer uso de otra de las ventajas importantes de los diseños orientados a objetos, el *polimorfismo*. Es decir, se podría solicitar la ejecución de este método en uno de los objetos instanciados, simplemente especificando el tipo de objeto a partir de una referencia de la clase padre. De esta manera, enviando el mismo mensaje (se solicita la ejecución de la operación) a partir de una referencia, el objeto que lo reciba ejecutará dicha operación de acuerdo a su propia implementación.

En la figura 3 se establecen dos formas distintas de llevar a la práctica esta propiedad polimórfica de los diseños orientados a objetos. Por un lado, el uso de métodos *abstractos* en clases base de la derivación (hablaríamos de clases *abstractas*), y por otro, el empleo de *interfaces* que declaran también una serie de métodos virtuales. El uso de métodos abstractos (vacíos de código) fuerza a que las clases derivadas que quieran emplearlos deban implementar su codificación; por tanto, en este caso no se habla de *redefinir* los métodos, pues no hay definición alguna en la clase padre. Sin embargo, tal como se indica en el párrafo anterior, se podría contemplar una clase base no abstracta y redefinir sus métodos (no abstractos) en clases derivadas. El inconveniente en este caso, es que la

implementación de los métodos de la clase base quedaría anulada cuando se emplee el polimorfismo.

Veamos un ejemplo concreto de uso de métodos abstractos, a partir del diseño mostrado en la figura anterior (figura 3). Se plantea la declaración de una referencia a la clase *Geometria* (clase base abstracta) que podríamos llamar *geom1* y otra referencia de la misma clase a la que llamaremos *geom2*. A continuación, podemos crear un objeto de la clase *RectanguloGrafico* asociado a la referencia *geom1* y creamos otro objeto de la clase *CirculoGrafico* que se asociará a la referencia *geom2*. Como vemos, estas dos clases derivan de las clases *Rectangulo* y *Circulo*, respectivamente, las cuales a su vez derivan de la clase base *Geometria*. Pues bien, con estas especificaciones podemos referirnos a un objeto *CirculoGrafico* a partir de la referencia *geom2*, teniendo en cuenta que podemos enviar a dicho objeto un mensaje del tipo *geom2.perimetro( )* o del tipo *geom2.area( )*. Ambos métodos pertenecen a la clase base, pero lógicamente se ejecutará la implementación de estas operaciones en su clase derivada *CirculoGrafico*. Lo mismo ocurrirá si enviamos estos dos mensajes a un objeto *RectanguloGrafico* (empleando, en este caso la referencia *geom1*, lógicamente). Por tanto, queda claro que el objeto que reciba el mensaje responderá al mismo de acuerdo a su comportamiento definido.

Sin embargo, se puede observar que la implementación de los métodos *perimetro( )* y *area( )* no se lleva a cabo en las clases *RectanguloGrafico* y *CirculoGrafico*, de hecho, ni siquiera aparecen como operaciones en el diseño de las mismas. Entonces, ¿cómo es posible que puedan emplear sus objetos dichos métodos?. Lógicamente, la respuesta está en la relación de herencia que se establece entre dichos objetos y los objetos *Rectangulo* y *Circulo*, respectivamente, los cuales si recogen e implementan estas operaciones. En el diseño de las clases no se muestra explícitamente los atributos y métodos heredados de sus clases padre, pero precisamente se establece esta relación de herencia para beneficiarse de las propiedades y comportamiento de éstas. Se observa, por tanto, el potencial que alcanza el *polimorfismo* en combinación con la *herencia*.

Como se ha dicho, una segunda forma de hacer uso del polimorfismo en este diseño es mediante el uso de interfaces. Concretamente, se ha diseñado la interfaz *Dibujable* con esa finalidad. De igual forma que en el caso anterior, se puede crear un objeto de la clase

CirculoGrafico y ligarlo a una referencia de la interfaz que implementa (por ejemplo, la referencia *dib2*). Esta referencia servirá para enviar un mensaje al objeto en cuestión, que actuará de acorde a la codificación del método de la interfaz que recoge. En definitiva, si enviamos el mensaje *dib2.dibujar( )* se dibujará un círculo, a partir de los valores que tengan los atributos que recogen esa información en el objeto.

Por tanto, en el contexto del polimorfismo, las referencias de tipo interfaz se pueden utilizar de modo similar a las referencias ligadas a clases abstractas. De esta manera, el empleo de referencias de una clase base o de una interfaz permite tratar de un modo unificado objetos distintos, pertenecientes a distintas clases derivadas de una clase base o bien a clases distintas que implementan esta interfaz. Sin embargo, el uso de las interfaces permite abstraer los métodos a un nivel superior. Una interfaz eleva el concepto de clase abstracta hasta su grado más alto, puede verse como una forma, un molde, que proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia. En este sentido, las interfaces permiten ampliar aún más las posibilidades del polimorfismo, por su mayor flexibilidad y por su independencia de la jerarquía de clases del diseño.

Una limitación clara de los programas no orientados a objetos, que no permite el uso del polimorfismo, es la necesidad de reescribir el código fuente para contemplar nuevas posibilidades. Tradicionalmente, ante diversas alternativas A, B, C para proceder en una situación se organiza una estructura *CASO* del tipo:

```
Selección de caso
        A: acción_si_A;
        B: acción_si_B;
        C: acción_si_C;
...
```

En nuestro ejemplo, la acción a llamar podría ser la función *dibujar*, que estaría codificada de manera distinta en cada caso (cada opción representaría una figura geométrica concreta). Observamos en esta estructura cómo las funciones adoptan un papel principal sobre los datos (*dibujar_si_X*), y cómo se evalúa de manera secuencial la ejecución de una función u otra; típico de la programación estructurada. Si en un momento dado aparece una nueva opción D no queda más remedio que retocar la estructura *CASO* para añadirla, con el

posible agravante de no disponer del código fuente del programa para llevar a cabo esta modificación.

En cambio, en un programa orientado a objetos, no es necesario utilizar esta estructura, sino definir adecuadamente el método para cada clase diferente (tipo de figura geométrica), tal como se ha venido explicando. Además, un sistema orientado a objetos puede crecer sin necesidad de que el programador conozca los detalles de implementación (código fuente) de los elementos que permiten el funcionamiento de dicho sistema.

Continuando con el ejemplo, cuando se invoque el método *dibujar* haciendo uso del polimorfismo (*X.dibujar*), el objeto X responderá de una manera u otra según la clase a la que pertenezca. En este caso, los datos adoptan un papel principal sobre las funciones (métodos): se le indica al objeto X que "se dibuje". Si se desea modificar su funcionalidad, basta derivar una nueva clase y sobreescribir el método *dibujar* en dicha clase, sin que sea necesario conocer el código fuente de este método en la clase padre, únicamente se requiere tener en cuenta su *signatura* (nombre, tipo y número de argumentos). La facilidad de extensión de este sistema salta a la vista, no se necesita para nada el código fuente de los elementos del sistema, únicamente necesitamos conocer los comportamientos (métodos) que queremos sobreescribir, pero no su implementación, que precisamente se encuentra encapsulada. El encapsulado de la información, como se dijo en su momento, permite ocultar a los usuarios del sistema la propia programación de un objeto, ofreciéndoles únicamente una interfaz externa mediante la cual interactuar con el objeto.

El polimorfismo, como elemento clave en la programación orientada a objetos, junto a los conceptos también ejemplificados de herencia y sobrecarga de métodos, son tres características principales en el desarrollo de software reutilizable.

Por último, hay que hacer hincapié en que un diseño bien establecido y planteado desde un principio para que tenga en cuenta las características propias del lenguaje OO que se empleará en su codificación, lógicamente facilitará la implementación definitiva del código. En este sentido, el diseño orientado a objetos de nuestro ejemplo (figura 3) recoge algunas de las particularidades propias del lenguaje de programación *Java*. Por ejemplo, utilizamos su método *super( )*, que incluimos en el constructor *CirculoGrafico( )* de la clase *CirculoGrafico*, para llamar al constructor de su clase padre. Además, hemos

aprovechado la capacidad que posee este lenguaje para manejar tipos de datos primitivos (*double*, *int*) en la especificación de las características de los atributos, parámetros y valores de retorno que se manejan en el diseño. Otro elemento a destacar, la palabra *void*, se usa para indicar que el método no tiene valor de retorno, esto es, que no devuelve ningún valor al objeto que solicitó la ejecución de dicho método.

3. FASES DEL MODELADO ORIENTADO A OBJETOS

En el proceso de modelado orientado a objetos (OO) podemos distinguir las tres fases tradicionales de elaboración de un sistema informático (Pressman, 1993):

1) *Análisis*: se ocupa del qué, de entender el dominio del problema.
2) *Diseño*: responde al cómo, y se centra en el espacio de la solución.
3) *Implementación*: fase en la que se adapta la solución a un entorno de programación específico.

Básicamente, los productos del análisis orientado a objetos (AOO) sirven como modelos de los que se puede partir para un diseño orientado a objetos (DOO); los productos del DOO pueden utilizarse entonces como anteproyectos para la implementación completa de un sistema utilizando métodos de programación orientada a objetos (POO).

En este contexto de la OO, la transición entre los modelos de análisis y diseño se vuelve más fácil e intuitiva, puesto que los objetos del dominio del problema tienen una correspondencia con los objetos del dominio de la solución. Como señalan McGregor y Korson (1990), durante el análisis se abordan los objetos del dominio del problema, mientras que en el diseño se especifican los objetos adicionales necesarios para obtener la solución implementada en el ordenador. Precisamente la no consideración de ambas fases ha conducido al desastre a no pocos proyectos de software.

Se han propuesto un gran número de métodos para el análisis y diseño orientado a objetos, como el OOSD de Wasserman, Pircher y Muller (1990), el método de Coad y Yourdon (1991a, 1991b), el OODLE de Shlaer y Mellor (1991), el método de Desfray (1992), el método de Booch (1991, 1996), el OMT de Rumbaugh, Blaha, Premerlani, Eddy y Lorensen (1991), el OOSE de Jacobson, Christerson, Jonsson y Övergaard (1992), el método de Embley, Kurtz y Woodfield (1992), el HOOD de Robinson (1992), el método Petch de Martín y Odell (1994), el SOMA de Graham (1994a), el método Fusion de Coleman et al. (1994), etc.

Cabe destacar el método de Booch (1991, 1996), el OMT de Rumbaugh et al. (1991), y el OOSE de Jacobson et al. (1992). Booch, Rumbaugh y Jacobson unificaron la notación de sus respectivos métodos en 1996, dando lugar al lenguaje de modelado conocido como UML (Booch et al., 1999), que sufrió un proceso de estandarización en 1997 por el *Object*

Management Group (OMG), cuyas especificaciones han sido refrendadas por la mayoría de las empresas e instituciones desarrolladoras de software. El OMG es un consorcio a nivel internacional que integra a los principales representantes de la industria de la tecnología de información OO, y que tiene como objetivo central la promoción, fortalecimiento e impulso de la industria OO. Para ello, propone y adopta por consenso especificaciones que afectan a la tecnología orientada a objetos, como es el caso de la especificación UML, que junto con el *Proceso Unificado de Desarrollo de Software* (*véase* apartado 4) están consolidando esta tecnología OO.

La motivación de estos tres autores por crear un lenguaje unificado de modelado (UML) se fundamentó en tres razones (Booch et al., 1999). En primer lugar, cada uno de los métodos ya esta evolucionando independientemente hacia los otros dos. Tenía sentido hacer continuar esa evolución de forma conjunta en vez de hacerlo por separado, eliminando la posibilidad de cualquier diferencia gratuita e innecesaria que confundiera aún más a los usuarios. En segundo lugar, al unificar estos métodos pretendían proporcionar cierta estabilidad al mercado orientado a objetos, permitiendo que los proyectos se pusieran de acuerdo en un lenguaje de modelado maduro y permitiendo a los constructores de herramientas que se centraran en proporcionar más características útiles. En tercer lugar, los autores esperaban que su colaboración introduciría mejoras en los tres métodos anteriores, ayudándose entre ellos a capturar lecciones aprendidas y a cubrir problemas que ninguno de sus métodos anteriores había manejado bien anteriormente.

El hecho, es que cada uno de éstos era un método completo, aunque todos tenían sus puntos fuertes y débiles. En resumen, el método de Booch era particularmente expresivo durante las fases de diseño y construcción de los proyectos, OOSE proporcionaba un soporte excelente para los *casos de uso* como forma de dirigir la captura de requisitos, el análisis y diseño de alto nivel, y OMT-2 era principalmente útil para el análisis y para los sistemas de información con gran cantidad de datos.

Con posterioridad al desarrollo de UML, los tres autores presentaron el *Proceso Unificado de Desarrollo de Software* (Jacobson et al., 2000), que también se denominado de forma abreviada *Proceso Unificado* (PU), como un estándar que cubre todo el proceso ligado al ciclo de vida de desarrollo de software. De hecho, el PU integra una amplia variedad de

aportaciones, no sólo las proporcionadas por los procesos involucrados en los propios métodos de sus autores (párrafo anterior). La notación UML se integra dentro de este proceso, puesto que permite modelar de forma gráfica los diferentes elementos involucrados en el desarrollo de software.

Actualmente, UML posee una gran relevancia como lenguaje de modelado en el desarrollo de software OO gracias a su nivel de estandarización, aspecto éste que permite una mayor y mejor comunicación entre desarrolladores de software.

A grandes rasgos y salvando las diferencias que existen entre los distintos métodos a los que hemos hecho referencia al principio de este apartado, sus etapas se pueden sintetizar de la siguiente manera:

1) Se identifican las clases semánticas (con sus atributos y servicios), esto es, las que reflejan el espacio del problema. Para ello, la mayor parte de los métodos proponen un enfoque «lingüístico» mediante el análisis de las oraciones contenidas en la especificación inicial del modelo (Abbott, 1983):

    - Los sustantivos se consideran clases candidatas.
    - Los verbos se consideran posibles servicios o métodos de las clases.
    - Los adjetivos a menudo expresan atributos de la clase, etc.

    Antes de proceder al análisis lingüístico, es recomendable extraer las frases nominales cambiando los plurales por singulares, unificando los términos, etc. (Peralta y Rodríguez, 1994).

    También se propone identificar las clases partiendo de los *guiones* (Jacobson et al., 1992), de las listas de comprobación, de los análisis de dominio, o utilizando *patrones* (Coad, 1992), *layers* (Graham, 1994a), etc.

2) Se establecen las interrelaciones entre las clases, sobre todo, estudiando la generalización (herencia) y la agregación entre las mismas.

3) Se añaden otras clases, como las de interfaz de usuario, las que incorporan los mecanismos de control de la aplicación y las clases base, que son independientes de la aplicación y que generalmente interactúan con el sistema operativo propietario.

4) Se implementan las clases.

En cuanto al paso de la fase de análisis a la de diseño, se pueden distinguir dos aproximaciones:

- De *elaboración*: el modelo de análisis se va completando con información de diseño hasta que se encuentre listo para ser codificado.
- De *transformación*: se aplican reglas para transformar el análisis en diseño.

Es importante resaltar el hecho de que las fronteras entre análisis y diseño son difusas, aunque el objetivo principal de ambos es bastante diferente. En el análisis orientado a objetos (AOO) se persigue modelar el mundo a partir de las clases y objetos que forman el vocabulario del dominio del problema, y en el diseño (DOO) se establecen las abstracciones y mecanismos que proporcionan el comportamiento que este modelo requiere. La existencia de estas fronteras difusas entre la fase de análisis y la de diseño es un aspecto que deriva de la forma en que se afronta el desarrollo de software OO. La aparición de los lenguajes de programación orientados a objetos ha hecho que se tienda a cambiar el paradigma de desarrollo empleado hasta el momento (el llamado *ciclo de vida clásico*) en la creación de programas.

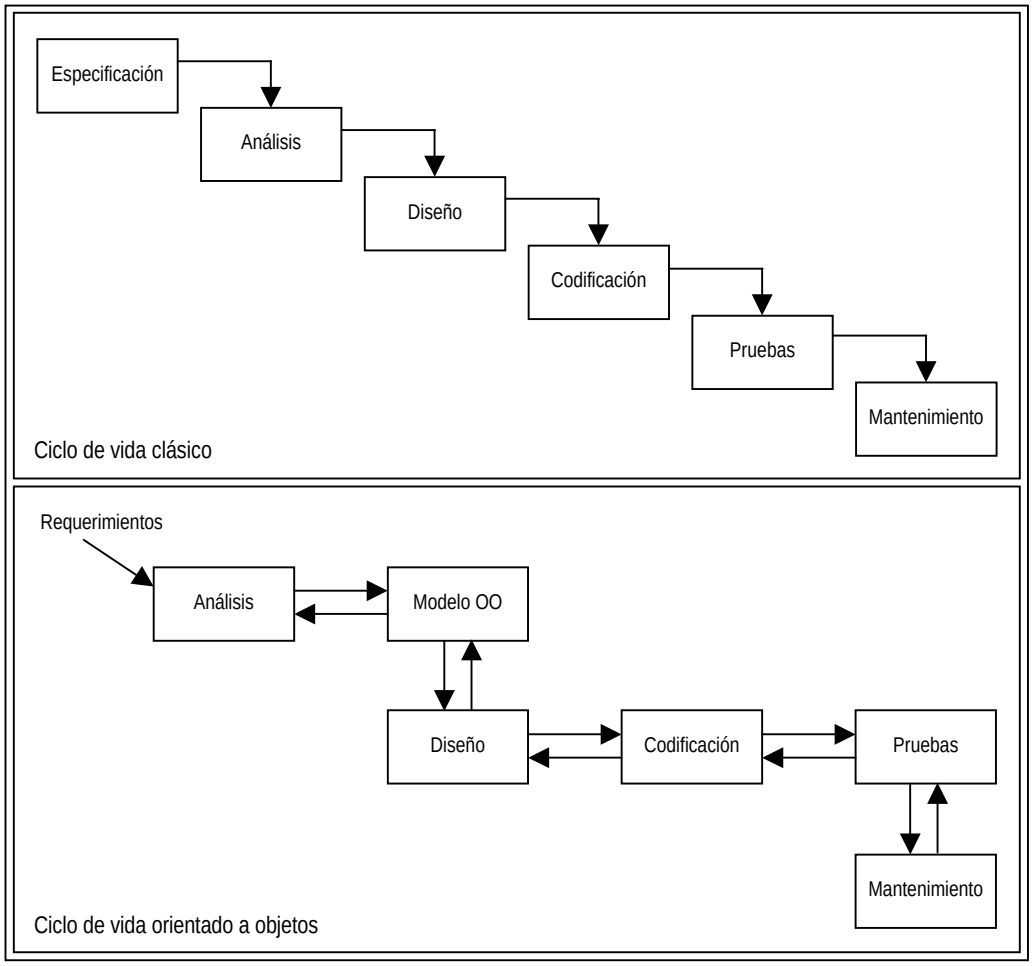


Figura 4. Los ciclos de vida clásico y orientado a objetos
(Fuente: Sánchez, 1996)

En el *ciclo de vida clásico* (figura 4), en cada etapa del proceso se generan unos nuevos productos que son especificaciones de lo que se ha de hacer en la etapa siguiente. Este paradigma se ha sustituido por otros, como el *ciclo de vida para orientación a objetos* (Alfonseca y Alcalá, 1992), más iterativos, que permiten y potencian que en diversas etapas del proceso se vaya hacia delante y hacia atrás; en palabras de Gilb (1990, p. 24), se trata de *«(...) un poco de análisis, un poco de diseño, un poco de programación, y vuelta a empezar»*. En el capítulo 4 nos detenemos en el ciclo de vida OO propuesto por Jacobson et al. (2000), antes anunciado como el *Proceso Unificado de Desarrollo de Software*.

Como se observa en la figura 4, el diagrama del *ciclo de vida orientado a objetos* muestra este proceso iterativo que se mueve hacia delante y hacia atrás. En ese sentido, en el

diagrama en cuestión los diversos modelos que forman el *Modelo OO* se van construyendo o rehaciendo a medida que el proceso transcurre desde el análisis de requerimientos hasta la implementación y uso del sistema.

En los años 70 y 80 aparecieron métodos de desarrollo estructurado asociados al ciclo de vida clásico. Sin embargo, los métodos de programación estructurada no se acoplan bien al paradigma orientado a objetos, tal como se discutió en su momento en el apartado 2.3.2. Por un lado, se centran en la descomposición algorítmica, es decir, en el proceso de transformación de los datos y no en los datos mismos. Además, tampoco están pensados para manejar el proceso de encapsulado de la información ni para tratar con jerarquías.

En consecuencia, de forma paralela a la adopción del paradigma orientado a objetos, han ido apareciendo métodos de análisis y diseño orientados a objetos pensados para permitir el desarrollo de software siguiendo este paradigma, bastantes de los cuales han sido mencionados al comienzo de este apartado. Cada uno de estos métodos dispone de un conjunto definido de entidades, símbolos gráficos, diagramas, fichas, reglas y procedimientos de actuación. El objetivo último de todos ellos es la creación de sistemas software de calidad y, como es natural, sus metodologías no están aisladas ni son totalmente diferentes las unas de las otras, sino más bien al contrario. Un estudio comparativo de los principales métodos –en número superior a treinta– lo encontramos en de Champeaux y Faure (1992), en el que se analizan los rasgos comunes y sus diferencias.

En los siguientes subapartados nos centramos en el análisis orientado a objetos (AOO), el diseño orientado a objetos (DOO), así como en la fase de implementación o codificación del diseño mediante la programación orientada a objetos (POO).

3.1. ANÁLISIS

Siguiendo a Booch (1996), el análisis orientado a objetos (AOO) se puede definir como «un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema».

Este mismo autor, examina varios enfoques contrastados para el análisis que son de relevancia para los sistemas orientados a objetos: enfoques clásicos, enfoques de análisis del comportamiento, enfoques de análisis del dominio, análisis de casos de uso, fichas CRC y descripción informal en lenguaje natural.

Enfoques clásicos. Estos enfoques derivan sobre todo de los principios de la categorización clásica. Por ejemplo, Shlaer y Mellor (1988) sugieren que las clases y objetos candidatos provienen habitualmente de una de las fuentes siguientes: cosas tangibles, papeles (roles), eventos e interacciones. Desde la perspectiva del modelado de bases de datos, Ross (1987) ofrece una lista similar: personas, lugares, cosas (objetos físicos o grupos de objetos tangibles), organizaciones, conceptos (principios o ideas no tangibles *per se*) y eventos.

Análisis del comportamiento. Mientras que los enfoques clásicos se centran en aspectos tangibles del domino del problema, los enfoques de análisis del comportamiento se centran en el comportamiento dinámico como fuente primaria de clases y objetos. Estos enfoques tienen más que ver con el agrupamiento conceptual: se forman clases basadas en grupos de objetos que exhiben comportamiento similar.
Autores como Wirfs-Brock, Wilkerson y Wiener (1990), por ejemplo, hacen hincapié en las responsabilidades, entendidas como «el conocimiento que un objeto tiene y las acciones que un objeto puede realizar. Las responsabilidades están encaminadas a comunicar una expresión del propósito de un objeto y su lugar en el sistema. Las responsabilidades de un objeto son todos los servicios que suministra para todos los contratos que soporta». De este modo, se agrupan cosas que tienen responsabilidades comunes, y se forman jerarquías de clases que involucran a superclases que incorporan responsabilidades generales y subclases que especializan su comportamiento.

Análisis de dominios. El análisis de dominios busca identificar las clases y objetos comunes a todas las aplicaciones dentro de un dominio dado, mientras que los enfoques anteriores se aplican típicamente al desarrollo de aplicaciones concretas e independientes. Si uno está a mitad de un diseño y le faltan ideas sobre las abstracciones clave que existen, un pequeño análisis del dominio puede ayudarnos indicándonos las abstracciones clave que han demostrado ser útiles en otros sistemas relacionados con el nuestro. Se define el

análisis de dominios como «un intento de identificar los objetos, operaciones y relaciones que los expertos del dominio consideran importantes» (Arango, 1989, p.153). Moore y Bailin (1988) sugieren los siguientes pasos en el análisis de dominios:

- Construir un modelo genérico informal del dominio consultando con expertos de ese dominio.
- Examinar sistemas existentes en el dominio y representar esta comprensión en un formato común.
- Identificar analogías y diferencias entre los sistemas consultando con expertos del dominio.
- Refinar el modelo genérico para acomodar sistemas ya existentes.

Un experto del dominio habla el vocabulario del dominio del problema, es decir, es una persona profundamente familiarizada con todos los elementos de un problema particular. En este sentido, para aclarar un problema de diseño todo lo que se necesita normalmente es una reunión entre un experto del dominio y un desarrollador. Por otra parte, hay que decir que el análisis de dominios nunca es una actividad monolítica, está mejor enfocado si se alterna el análisis con el diseño: se analiza un poco y después se diseña otro poco, tal como se afronta el ciclo de vida de desarrollo de software en la OO (*véase* figura 4).

Análisis de casos de uso. La práctica en el análisis de casos de uso se puede acoplar con los tres enfoques anteriores (análisis clásico, análisis del comportamiento, análisis de dominios), con la finalidad de dirigir el proceso de análisis de modo significativo. De esta manera, se reduce el riesgo asociado al uso aislado de estos tres enfoques en ese proceso de análisis, que exige una gran cantidad de experiencia personal por parte del analista. El análisis de casos de uso ha sido formalizado originalmente por Jacobson et al. (1992), que definen un *caso de uso* como «una forma o patrón o ejemplo concreto de utilización, un escenario que comienza con algún usuario del sistema que inicia alguna transición o secuencia de eventos interrelacionados». Jacobson, como integrante del grupo de diseño de UML y del PU, aportó a este estándar de modelado un soporte excelente para los casos de uso como forma de dirigir la captura de requisitos, el análisis y el diseño de alto nivel. Se analizará el empleo de casos de uso en el cuarto capítulo, dedicado al *Proceso Unificado de Desarrollo de Software* (Jacobson et al., 2000).

Fichas CRC. Las fichas CRC han surgido como una forma simple, pero muy efectiva de analizar escenarios. Propuestas en primer lugar por Beck y Cunningham (1989) como una herramienta para la enseñanza de programación, las fichas CRC han demostrado ser una herramienta de desarrollo muy útil que facilita las «tormentas de ideas» y mejora la comunicación entre los desarrolladores. Una ficha CRC no es más que una tarjeta sobre la cual el analista escribe el nombre de una clase (en la parte superior de la tarjeta), sus responsabilidades (en una mitad de la tarjeta) y sus colaboradores (en la otra mitad). Se crea una ficha para cada clase que se identifique como relevante para el escenario. Estas fichas pueden disponerse espacialmente para representar patrones de colaboración.
En el capítulo 2 de este trabajo se hizo referencia a un diseño específico de ficha o tarjeta CRC (apartado 2.3.4.b, figura 2) presentado por Graham (1994b) y en la que se emplea la notación SOMA propuesta por el mismo autor (Graham, 1994a).

Descripción informal en lenguaje natural. Una alternativa radical para el análisis orientado a objetos clásico fue propuesta primeramente por Abbott (1983), quien sugirió redactar una descripción del problema (o parte del problema) en lenguaje natural, y subrayar entonces los nombres y los verbos. Los nombres o sustantivos representan objetos candidatos, y los verbos representan operaciones candidatas sobre ellos. Los adjetivos encontrados en ese análisis «lingüístico» del problema pueden expresar atributos de los objetos.
El enfoque de Abbott es útil porque es simple y porque obliga al desarrollador a trabajar en el vocabulario del espacio del problema. Sin embargo, de ninguna manera es un enfoque riguroso, y desde luego no se adapta bien a los problemas complejos. El lenguaje humano es un vehículo de expresión bastante impreciso, así que la calidad de la lista resultante de objetos y operaciones depende de la habilidad del autor para redactar. Además, todo nombre puede transformarse en verbo, y al revés; por tanto, es fácil dar un sesgo a la lista de candidatos para enfatizar bien los objetos o bien las operaciones (Booch, 1996).

Queda claro, por tanto, que el AOO persigue modelar el mundo a partir de las clases y objetos que forman el vocabulario del dominio del problema, y que existen diferentes métodos para analizar en esos términos de orientación a objetos los requisitos del sistema del que se extrae el problema.

Si se analizan los problemas que se enuncian en el marco de la Estadística, se observa que son los datos el elemento principal a destacar de dichos problemas, y que queda en un segundo plano de estudio lo se haga con ellos, aspecto que depende entre otros factores de las características de estos datos. Esta perspectiva, que otorga a los datos un papel de mando sobre las funciones, se corresponde con el paradigma OO más que con el paradigma algorítmico (*véase* apartado 2.3.2). En definitiva, una clase son unos datos y unos métodos que operan sobre esos datos.

Precisamente, este papel preponderante que se otorga a los datos en el contexto de los problemas estadísticos permite que el analista, diseñador y programador estadístico pueda beneficiarse de las ventajas de la orientación a objetos (*véase* apartado 2.3.4).

Por un lado, gracias a la encapsulación es posible desarrollar por separado y de manera independiente los tipos y estructuras de datos necesarios para cada problema concreto dentro del sistema; de esta manera se evita que un cambio en los detalles internos (ocultos) de una clase afecte al resto del sistema.

Por otra parte, la herencia es una propiedad que puede facilitar mucho la programación, ya que muchos estadísticos difieren únicamente en pequeños detalles (por ejemplo, la variancia muestral y la variancia muestral corregida): cada estadístico (objeto) sería dispuesto en una jerarquía en la que únicamente se modificarían los detalles oportunos del objeto de interés, conservando los elementos heredados (atributos y comportamientos) del resto de objetos jerárquicamente superiores.

Gracias al polimorfismo, se podrían enviar mensajes del tipo “calcúlate” o “represéntate gráficamente” a diferentes estadísticos (objetos). Cada uno de estos objetos respondería al mismo mensaje en función de la implementación o codificación realizada sobre el método del objeto receptor que es capaz de entender el mensaje; en otras palabras, cada estadístico interpretaría de forma distinta las instrucciones (mensaje) que reciba.

En definitiva, este primer análisis general de las ventajas del paradigma OO en el contexto estadístico, parece indicar en principio que la orientación a objetos es un buen “caldo de cultivo” para fomentar la idoneidad del uso de herramientas estadísticas modernas que recojan estas propiedades de la OO, y de esta manera, comenzar a activar el tránsito “de usuario a programador” del que hablaba Chambers (2000).

3.2. DISEÑO

El objetivo de la fase de diseño es la generación de una descripción sobre cómo sintetizar los objetos extraídos del dominio de la solución al problema, cuyo comportamiento esté de acuerdo con el modelo de análisis y con el resto de requisitos del sistema asociados a la fase de implementación de dichos objetos como componentes de software. En definitiva, el principal resultado del diseño es el modelo de diseño, que se esfuerza en conservar la estructura del sistema impuesta por el modelo de análisis, y que sirve como esquema para la implementación.

Los analistas ven los objetos como descripciones de propiedades y comportamientos requeridos. Los diseñadores, en cambio, ven estos objetos como entidades computacionales que llevan a efecto o realizan estos comportamientos y propiedades de una manera que sea posible implementarlos en lenguajes de programación OO comunes. En este sentido, el diseño actúa como un puente entre las actividades de análisis que describen los requisitos o propiedades que el sistema debe tener, y las actividades de implementación que describen la plataforma de programación y condiciones propias del lenguaje en que se crearán los componentes físicos de dicho sistema.

De forma más abstracta, Booch (1996) define el diseño orientado a objetos como «un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseña».

Por supuesto, esta definición se encuentra ligada al propio método OO propuesto por Booch, particularmente expresivo durante la fase de diseño y uno de los más populares antes de su integración en UML y PU junto con los métodos OOSE (Jacobson et al., 1992) y OMT (Rumbaugh et al., 1991). El método de Booch (1996) recomienda el uso de cuatro modelos: lógico, físico, estático y dinámico. Concretamente, en el diseño del sistema considera la dimensión estática y dinámica mediante dos niveles para cada una de ellas: el nivel lógico y el nivel físico. De ahí los cuatro modelos enunciados.

1. El modelo *lógico* contempla las clases y objetos existentes y las relaciones que se establecen entre ellas.
2. El modelo *físico* se refiere a la arquitectura del sistema, organizada en módulos y en procesos.
3. El modelo *estático*, tanto lógico como físico, contempla los aspectos estructurales –estructura de clases y de módulos– y las relaciones –entre clases (herencia, asociación, agregación) y entre módulos (dependencias).
4. El modelo *dinámico* se refiere al comportamiento del sistema, tanto a nivel lógico (escenarios de utilización), como físico (asignación de procesos a módulos y procesadores).

La expresión de estos modelos se lleva a cabo mediante diversos diagramas:

1. *Diagrama de clases*: muestra la existencia de clases y las relaciones entre ellas en la perspectiva lógica y estática del sistema.
2. *Diagrama de objetos*: muestra la existencia de objetos y las relaciones entre ellos en la perspectiva lógica y dinámica (puesto que permiten reseguir la ejecución de un escenario) del sistema.
3. *Diagramas de transición de estados*: muestra el comportamiento dinámico de las clases describiendo los sucesivos estados en que pueden existir. Se refieren, por tanto, a la perspectiva lógica y dinámica del sistema.
4. *Diagramas de módulos*: muestra la asignación de clases a módulos en la perspectiva física y estática del sistema.
5. *Diagramas de procesos*: muestra la asignación de procesos a los procesadores en la perspectiva física y dinámica del sistema.

En concreto, el modelo de diseño que se presentó como ejemplo en el capítulo 2 (figura 3, apartado 2.3.5) fue expresado mediante un *diagrama de clases* (bajo notación UML). Por supuesto, son posibles otros diseños de este mismo problema, a partir de otros tipos de diagramas (diferentes puntos de vista). Por ejemplo, podría ser interesante plantear el mismo diseño desde la perspectiva de la interacción entre objetos (instancias concretas de los elementos encontrados en el diagrama de clases). En este sentido, y también bajo

especificación UML (*véase* capítulo 5), haríamos uso del llamado *diagrama de objetos*, que expresaría la parte estática de una interacción, que consiste en los objetos que colaboran, pero sin ninguno de los mensajes enviados por ellos. Este diagrama de objetos congela un instante en el tiempo[1]. En UML, para poder modelar la interacción dinámica entre objetos durante la ejecución del programa haríamos uso de los *diagramas de interacción*. Este tipo de diagramas muestran una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar entre ellos. Podemos diferenciar entre *diagramas de secuencia*, que son diagramas de interacción que destacan la ordenación temporal de los mensajes, y *diagramas de colaboración*, en los que destaca la organización estructural de los objetos que envían y reciben mensajes (Booch et al., 1999).

Una última referencia a la etapa de diseño nos lleva a hablar de los *patrones de diseño*, como una solución apropiada a un problema común en un contexto dado. El modo en que se suele entender actualmente este concepto fue acuñado por Gamma, Helm, Johnson y Vlissides (2003), importante referencia que se suele designar de forma abreviada como *GoF* (*Gang of Four*, "la banda del cuatro"). Los patrones de diseño cambian la perspectiva acerca de las posibilidades de diseño de la solución de un problema, permitiendo que estos diseños sean más flexibles, modulares, reutilizables y comprensibles.

A pesar de que este concepto incluye el término «diseño» y que ha sido incluido en este apartado, no se suele considerar que se refiera únicamente a la fase de diseño de un programa, es una solución completa que incluye análisis, diseño e implementación (Ocaña y Sánchez, 2003). De hecho, cada patrón de diseño se centra en un problema concreto, describiendo cuándo aplicarlo y si tiene sentido hacerlo teniendo en cuenta otras restricciones de diseño, así como las consecuencias y las ventajas e inconvenientes de su uso.

En general, un patrón tiene cuatro elementos esenciales (Gamma et al., 2003):

[1] A diferencia del diagrama de objetos del método de Booch, el diagrama de objetos de la especificación UML no contempla la perspectiva dinámica de interacción entre objetos; para ello, se debe acudir a los llamados *diagramas de interacción*.

1. El **nombre del patrón** permite describir, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias. Al dar nombre a un patrón estamos incrementando nuestro vocabulario de diseño, lo que permite diseñar con mayor abstracción.
2. El **problema** describe cuándo aplicar el patrón. Explica el problema y su contexto. Puede describir problemas concretos de diseño, así como las estructuras de clase u objetos que son sintomáticas de un diseño inflexible. A veces el problema incluye una serie de condiciones que deben darse para que tenga sentido aplicar el patrón.
3. La **solución** describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o una implementación en concreto, sino que un patrón es más bien como una plantilla que puede aplicarse en muchas situaciones diferentes. El patrón proporciona una descripción abstracta de un problema de diseño y cómo lo resuelve una disposición general de elementos (en nuestro caso, clases y objetos).
4. Las **consecuencias** son los resultados así como las ventajas e inconvenientes de aplicar el patrón. Aunque cuando se describen decisiones de diseño muchas veces no se reflejan sus consecuencias, éstas son fundamentales para evaluar las alternativas de diseño y comprender los costes y beneficios de aplicar el patrón. Las consecuencias de un patrón incluyen su impacto sobre la flexibilidad, extensibilidad y portabilidad de un sistema.

En resumen, los patrones de diseño se pueden entender como descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto. Los patrones de diseño ayudan a un diseñador a lograr un buen diseño más rápidamente.

Ocaña y Sánchez (2003), afirman que muchos de los patrones de diseño conocidos son directamente aplicables al diseño de programas estadísticos. De hecho, plantean una discusión acerca del uso de diferentes patrones en la resolución de problemas extraídos directamente del contexto de la simulación estadística. Proponen, por ejemplo, el empleo del patrón *Estrategia* (*Strategy* en *GoF*) en el diseño de un modelo OO que permita la

elección dinámica (en tiempo de ejecución) de diferentes algoritmos de generación aleatoria de variables bajo una determinada distribución.

El patrón *Estrategia* define una familia de algoritmos alternativos (estrategias distintas), encapsulados en una jerarquía de clases independiente de la jerarquía de clases clientes de estos algoritmos. Dichos algoritmos (clases) deben ser capaces de responder a un determinado mensaje implementado en un método adecuado; para ello, todos los algoritmos deben responder a una misma interfaz que proporcione dicho método. Las clases que van a ser "clientes" de estos algoritmos deben tener en su definición un atributo que sea una referencia al objeto algoritmo concreto que en un momento dado emplee ese cliente. La clase cliente dispondrá de un método que en general se limitará a activar el método del objeto algoritmo que en aquel momento tenga asignado como activo.

En el contexto del ejemplo mencionado, las clases "cliente" acuden a un algoritmo de generación aleatoria concreto (clase algoritmo) para generar una distribución de datos determinada (figura 5).

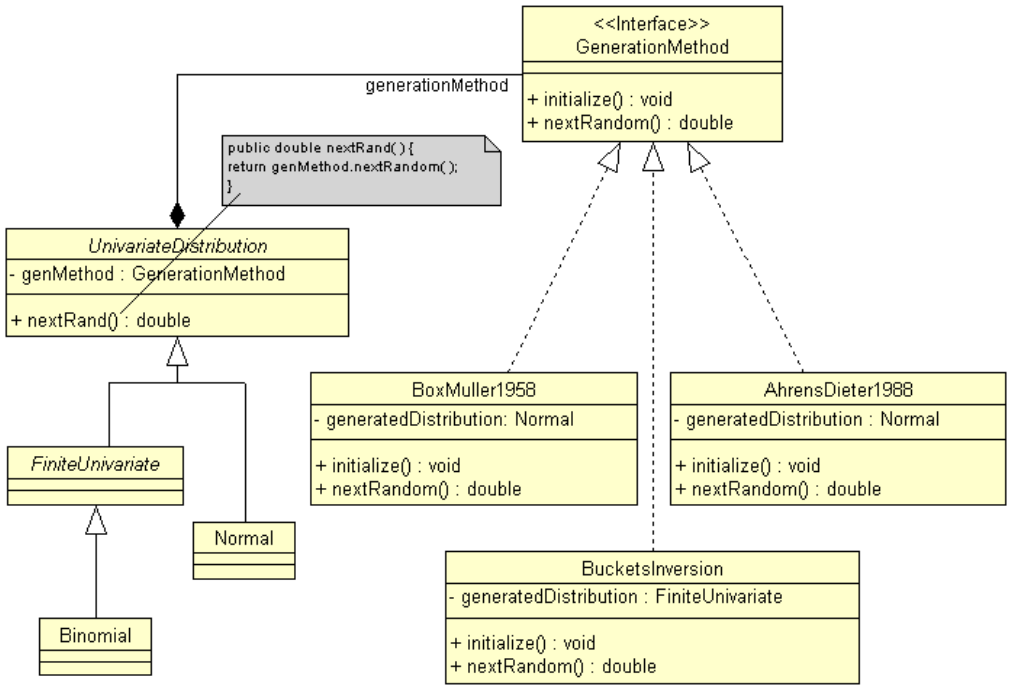


Figura 5. El patrón *Estrategia* en la generación aleatoria
(Fuente: Ocaña y Sánchez, 2003)

La clase abstracta *UnivariateDistribution* representa la clase base de la jerarquía de clases cliente. De ésta derivan la clase también abstracta *FiniteUnivariate* (de una clase abstracta no es posible generar objetos) y la clase *Normal*. A su vez, de la clase *FiniteUnivariate* deriva la clase *Binomial*. Gracias a la herencia, basta que la clase base de esta jerarquía tenga definida la referencia *genMethod* a los objetos que implementan algoritmos de generación (*AhrensDieter1988, BucketsInversion,...*), y que además tenga definido el método *nextRand( )*, que activa el algoritmo. Esta referencia está ligada a la interfaz *GenerationMethod*, de forma que los objetos que implementen dicha interfaz sean capaces de responder al mensaje *nextRandom* emitido por los objetos cliente (distribuciones); ello es posible gracias a la implementación del método *nextRandom( )* en las clases algoritmo. Cada objeto algoritmo responderá al mismo mensaje de manera distinta, a partir de su implementación concreta en el método asociado.

Veamos más en detalle este proceso. La implementación del método *nextRand( )* de la clase abstracta *UnivariateDistribution* (*véase* nota en figura 5), permite que un objeto cliente envíe el mensaje *nextRandom( )* a cualquier objeto algoritmo que implemente la interfaz *GenerationMethod*. En este sentido, el método *nextRand( )* se limita a activar el método *nextRandom( )* del objeto algoritmo que en aquel momento tenga asignado como activo. La referencia *genMethod* como tipo *GenerationMethod* es la que emplea precisamente el objeto cliente para enviar ese mensaje: *genMethod.nextRandom( )*. El objeto algoritmo que reciba el mensaje generará la distribución de valores a partir de la implementación propia de su método *nextRandom( )*. Una manera de controlar posibles errores de generación inadecuados para determinada distribución es precisamente que los algoritmos de generación tengan una referencia a la distribución concreta (al objeto) que están generando. Por ejemplo, se especifica que el algoritmo *AhrensDieter1988* solamente es válido para la clase *Normal* (o descendientes), mientras que el algoritmo *BucketsInversion* es válido para cualquier objeto de una clase descendiente de *FiniteUnivariate*.

La relación de «composición» entre un objeto compuesto (en este caso, una distribución) y una de sus partes (en este caso, un algoritmo de generación) se representa con una línea terminada con un rombo relleno, situado siempre en el lado del objeto compuesto (figura 5). La relación de composición es una variante de la relación de agregación entre dos

objetos. Una agregación (representada con rombo vacío) es un tipo de asociación entre dos clases u objetos conceptualmente situados a distinto nivel de importancia, una clase representa el "todo" (zona del rombo) y la otra representa una "parte" del todo. La composición también modela una relación "todo/parte", pero en este caso un objeto puede pertenecer sólo a una parte compuesta a la vez. Por ejemplo, si bien una *pared* puede ser parte de uno o más objetos *habitación* (relación de agregación simple), un *marco* pertenece exactamente a una *ventana* (relación de agregación compuesta). Además, en una relación compuesta, la parte compuesta es responsable de la disposición de las partes, lo que significa que debe gestionar la creación y destrucción de esas partes; siguiendo con el ejemplo, al crear un marco en un sistema de ventanas, debe asignarse a una ventana, y de forma análoga, al destruir la ventana, el objeto ventana debe a su vez destruir sus partes marco. En este sentido, volviendo al contexto de la figura 5, la duración del objeto que representa el algoritmo de generación ha de estar limitada por la duración del objeto distribución.

Es común que al mismo problema o situación sea aplicable más de un patrón de diseño, ya sea porque son posibles soluciones alternativas, o porque intervienen en partes distintas de la solución (Ocaña y Sánchez, 2003; Gamma et al., 2003). En el ejemplo que se está describiendo (figura 5) (Ocaña y Sánchez, 2003), es necesario que los objetos distribución y los objetos algoritmo de generación estén fuertemente vinculados, en concreto, que únicamente se pueda acceder al algoritmo de generación (incluyendo el hecho de crear el correspondiente algoritmo) desde el objeto distribución. En este sentido, no pueden existir algoritmos "libres" no asociados a una distribución. Esta necesidad conduce al uso de alguna de las variantes del patrón de diseño denominado comúnmente *Factoría* (*Factory*). De las variantes de este patrón, la más adecuada parece ser, según los autores de este ejemplo, la denominada *Factoría Polimórfica* (*Polymorphic Factory*), descrita en Eckel (2003). La implementación de este patrón permite crear objetos de una determinada clase simplemente llamando a un método de una clase general (la factoría, bajo control de las clases distribución en el ejemplo) y especificando el nombre de la clase deseada para el objeto, como información de entrada a dicho método.

Por supuesto, la discusión acerca de qué patrones de diseño pueden ser apropiados en la resolución de ésta y otras demandas estadísticas supone la necesidad de estar

familiarizados con los diversos patrones planteados en la literatura (Gamma et al., 2003; Eckel, 2003). Un análisis de las ventajas e inconvenientes de diseño de los patrones candidatos para resolver un determinado problema redundará en una mejor toma de decisiones; por ejemplo, la principal ventaja del patrón de diseño *Estrategia* es la mayor modularidad que proporciona, de forma que las clases correspondientes en el ejemplo a las distribuciones y las clases correspondientes a los algoritmos concretos de generación se pueden mantener y crecer de forma independiente. El principal inconveniente de este patrón, como el de la mayoría de patrones, es que su uso implica añadir un nivel de indirección, que puede suponer una cierta carga de computación adicional (Ocaña y Sánchez, 2003).

Otro ejemplo interesante, en el que los autores "redescubren" el patrón de diseño *Estrategia* (o se adelantan a *GoF*), se halla en Hitz y Hudec (1994). Estos autores plantean dicho patrón para la estimación del elipsoide de volumen mínimo en la determinación de la localización y la forma multivariante, donde el estimador de la localización es el centro del elipsoide de volumen mínimo que cubre al menos la mitad de los puntos de una muestra multivariante, y la matriz de covariancias representa la forma del elipse.

Otros patrones de diseño conocidos son, por ejemplo, el patrón *Adaptador* (*Adapter*), el patrón *Decorador* (*Decorator*), el patrón *Visitante* (*Visitor*), el patrón *Observador* (*Observer*), el patrón *Método Plantilla* (*Template Method*), el patrón *Constructor* (*Builder*), el patrón *Puente* (*Bridge*), etc. Siguiendo la nomenclatura y organización presentada por *GoF* (Gamma et al., 2003) en la clasificación de los distintos tipos de patrones, podemos hablar de patrones de creación (*Factory Method*, *Abstract Factory*, *Builder*, ...), patrones estructurales (*Adapter*, *Decorator*, *Bridge*, ...) y patrones de comportamiento (*Strategy*, *Observer*, *Template Method*, *Visitor*, *Iterator*, ...).

Los *patrones de creación* abstraen el proceso de creación de instancias, ayudan a hacer a un sistema independiente de cómo se crean, se componen y se representan sus objetos. Un patrón de creación de clases usa la herencia para cambiar la clase de la instancia a crear, mientras que un patrón de creación de objetos delega la creación de la instancia en otro objeto.

Los *patrones estructurales* se ocupan de cómo se combinan las clases y los objetos para formar estructuras más grandes. Los patrones estructurales de clases hacen uso de la herencia para combinar interfaces o implementaciones, aspecto particularmente útil para lograr que funcionen juntas bibliotecas de clases desarrolladas de forma independiente. Los patrones estructurales de objetos, en vez de combinar implementaciones o interfaces, describen formas de componer objetos para obtener una nueva funcionalidad, aspecto que otorga una flexibilidad añadida por la posibilidad de cambiar la composición en tiempo de ejecución, lo que es imposible con la composición de clases estática.

Los *patrones de comportamiento* tienen que ver con algoritmos y con la asignación de responsabilidades a objetos. Los patrones de comportamiento describen no sólo patrones de clases y objetos, sino también patrones de comunicación entre ellos. Estos patrones describen el flujo de control complejo que es difícil de seguir en tiempo de ejecución, lo que nos permite olvidarnos del flujo de control para concentrarnos simplemente en el modo en que se interconectan los objetos. Los patrones de comportamiento basados en clases usan la herencia para distribuir el comportamiento entre clases. Los patrones de comportamiento basados en objetos usan la composición de objetos en vez de la herencia. Un ejemplo de patrón de comportamiento basado en objetos es el patrón *Estrategia*, que como se ha visto permite encapsular un algoritmo en un objeto, facilitando la especificación y el cambio del algoritmo que usa el objeto compuesto.

Para finalizar, un último comentario en relación a la etapa de diseño, que se centra en los detalles finales del modelo desarrollado. Si bien un diseño representa una solución genérica para resolver el problema planteado en cualquier plataforma de programación, los detalles finales de ese diseño dependen precisamente del lenguaje de programación elegido para la implementación. En este sentido, tal como señalan Ocaña y Sánchez (2003), en C++ el elemento *GenerationMethod* del ejemplo expuesto será una verdadera clase, seguramente abstracta, y los algoritmos concretos deberían estar representados por clases que obligatoriamente desciendan de ella. En cambio, en lenguajes como Java hay más flexibilidad puesto que existe el concepto diferenciado de tipo interfaz, de manera que las clases “algoritmo” no deben pertenecer forzosamente a una misma jerarquía de herencia, basta con que implementen la interfaz *GenerationMethod* (como se muestra en la figura 5).

En el siguiente apartado se tratan, a nivel general, las implicaciones a tener en cuenta en la codificación del diseño final de un sistema bajo determinadas plataformas de programación, distinguiendo entre lenguajes orientados a objetos adecuados para este propósito y aquellos lenguajes que no cumplen con los requisitos marcados por el paradigma OO. Se enlaza, de manera específica, con la idoneidad de uso de plataformas de desarrollo OO en la implementación de soluciones estadísticas, tales como S-Plus o el entorno Omegahat.

3.3. IMPLEMENTACIÓN

La fase de implementación de la solución descrita en el diseño está condicionada por las características propias del lenguaje de programación OO elegido para la codificación. En la implementación codificamos el sistema en términos de componentes, es decir, ficheros de código fuente, ficheros de código binario, ejecutables y similares. Afortunadamente, la mayor parte de la arquitectura del sistema ha sido capturada durante el diseño, siendo el propósito principal de la implementación el desarrollar la arquitectura y el sistema como un todo a partir de la programación orientada a objetos (POO).

Booch (1996) define la POO como «un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia».

En la definición anterior hay tres partes importantes:

1. La programación orientada a objetos utiliza *objetos*, no algoritmos, como sus bloques lógicos de construcción fundamentales; desde el punto de vista físico, el bloque de construcción de los lenguajes orientados a objetos es una colección lógica de clases y objetos, en lugar de los subprogramas de la programación estructurada o algorítmica (*véase* apartado 2.3.2).
2. Cada objeto es una *instancia* de alguna clase.

3. Las clases están relacionadas con otras clases por medio de relaciones de *herencia* (la jerarquía de clases de la que se habló en el apartado 2.3.4).

Un programa (implementación final de un diseño) puede parecer orientado a objetos, pero si falta cualquier de estos elementos, no es un programa orientado a objetos. Según esta definición, por tanto, algunos lenguajes son orientados a objetos, y otros no los son. Stroustrup (1988) sugiere que «si el término 'orientado a objetos' significa algo, debe significar un lenguaje que tiene mecanismos que soportan bien el estilo de programación orientado a objetos... Un lenguaje soporta bien un estilo de programación si proporciona capacidades que hacen conveniente utilizar tal estilo. Un lenguaje no soporta una técnica si exige un esfuerzo o habilidad excepcionales escribir tales programas; en ese caso, el lenguaje se limita a permitir a los programadores el uso de esas técnicas». En este sentido, desde una perspectiva teórica, se puede fingir programación orientada a objetos en lenguajes de programación no orientados a objetos, como Pascal o incluso COBOL, pero no es deseable por lo complicada que puede llegar a ser esa tarea.

Por su parte, Cardelli y Wegner (1985) establecen los siguientes requisitos para considerar un lenguaje como orientado a objetos:

- Soporta objetos que son abstracciones de datos con una interfaz de operaciones con nombre y un estado local oculto.
- Los objetos tienen un tipo asociado (clase).
- Los tipos (clases) pueden heredar atributos de los supertipos (superclases).

Si un lenguaje no ofrece soporte directo para la herencia, entonces no es orientado a objetos. Estos autores distinguen tales lenguajes llamándolos *basados en objetos* en vez de *orientados a objetos*. Bajo esta definición, lenguajes como Smalltalk, Object Pascal, C++, Eiffel, CLOS y Java son todos ellos orientados a objetos, y Ada es basado en objetos. Sin embargo, al ser los objetos y las clases elementos de ambos tipos de lenguajes, es posible utilizar métodos de diseño orientado a objetos tanto para lenguajes orientados a objetos como para lenguajes basados en objetos. En este sentido, Stroustrup (1988) afirma que el uso del modelo OO durante la fase de diseño ayuda a explotar la potencia expresiva de los lenguajes de programación basados en objetos y orientados a objetos.

Retomando la sugerencia de Chambers (2000) acerca de la idoneidad de uso de herramientas estadísticas modernas que faciliten la extensión de la funcionalidad de sistemas en desarrollo, planteamos en este sentido el tránsito del usuario y desarrollador estadístico a plataformas concretas como S-Plus, R u Omegahat. Son herramientas estadísticas que cumplen con el listado de condiciones esbozadas por este mismo autor (Chambers, 2000) como requisitos básicos para el software de desarrollo. Estas condiciones ya fueron expuestas en el apartado introductorio de este trabajo, a saber: especificación fácil de tareas sencillas (1), capacidad de refinamiento gradual de tareas (2), posibilidades ilimitadas de extensión mediante programación (3), desarrollo de programas de alta calidad (4) y (5) posibilidad de integrar los resultados de los puntos 2 a 4 como nuevas herramientas informáticas.

Una de las virtudes de estas plataformas de desarrollo es que aportan una interfaz de usuario interactiva que permite la ejecución de tareas sencillas de una manera directa, a través de instrucciones interpretadas en una ventana de comandos; en otras palabras, las expresiones individuales redactadas por el usuario en un editor de comandos son leídas e inmediatamente ejecutadas. Las plataformas R, S-Plus y Omegahat hacen uso de los lenguajes interpretados R, S-Plus y Omegahat, respectivamente.

La ventaja de los lenguajes interpretados es que permiten un desarrollo incremental de la funcionalidad del sistema: un usuario puede escribir una determinada función, ejecutarla, volver a escribir otra función, ejecutarla, y entonces redactar una tercera función que llame a las dos funciones previas. De hecho, muchos usuarios comienzan a programar en estos entornos de trabajo, como S-Plus o R, sin apenas percatarse de que realmente están programando, si bien es cierto, utilizando para ello instrucciones de muy alto nivel y en un entorno interpretado.

En los lenguajes compilados (no interpretados), todas las instrucciones del programa son traducidas de forma conjunta por un compilador, que transforma el código fuente al lenguaje propio de la máquina que ejecutará el programa. Una vez que el programa está compilado, podrá ejecutarse independientemente del compilador. Los programas basados en lenguajes interpretados, sin embargo, necesitan en todo momento del programa intérprete para poder ser ejecutados, como ocurre con S-Plus, aunque con la ventaja

añadida de que permiten diseñar una interfaz interactiva cómoda para el usuario. En este sentido, lenguajes como Fortran, C, C++ e incluso Java fueron diseñados sin incluir de manera explícita al usuario (frente a la figura del programador) en su modelo computacional.

El lenguaje Java es un caso especial de lenguaje interpretado. Se enmarca dentro de la programación orientada a objetos, al igual que otros lenguajes como C++. Sin embargo, si bien C++ genera programas que requieren de un compilador, los programas Java son interpretados, aunque no en la misma forma que S-Plus. De hecho, el código fuente de un programa Java no es directamente interpretado, sino que previamente se traduce (compila) a un código intermedio llamado *bytecode*, que finalmente es interpretado por una máquina virtual (componente lógico) (Joyanes y Zahonero, 2002). La *Java Virtual Machine* (*JVM*) es la máquina virtual de Java que se encarga de interpretar y ejecutar una a una cada instrucción del programa. La ventaja principal de este sistema reside en que los programas pueden ser ejecutados en cualquier ordenador, independientemente de su arquitectura física y de su sistema operativo, simplemente es necesario que el ordenador disponga de dicha máquina virtual (JVM). Las implicaciones que tiene este sistema en relación a la distribución de software compatible con cualquier plataforma de procesamiento son obvias, puesto que permite compartir a través de la red cualquier componente implementado en Java. Por supuesto, al ser un lenguaje orientado a objetos no hay que perder de vista que la distribución de sus componentes (clases) abre las puertas a la ampliación de sistemas de una manera flexible.

Dentro del campo de la estadística, S-Plus combina su plataforma interactiva de lenguaje interpretado con la posibilidad de generar nuevos sistemas y ampliar los ya existentes. Para ello, recurre al lenguaje S-Plus, que a su vez se basa en el lenguaje S. La posibilidad de reutilizar software estadístico bajo esta plataforma con la finalidad de ampliar sistemas ha sido posible gracias a la evolución del lenguaje S en términos de funcionalidad orientada a objetos. Una guía del lenguaje S aplicado a la programación con datos se puede encontrar en Chambers (1988). Es una guía y referencia a la versión de S que subyace a la plataforma S-Plus en sus versiones 5.0 y superiores.

Otro entorno de desarrollo de software estadístico bastante extendido es el entorno y lenguaje de programación R (Ihaka y Gentleman, 1996). Al igual que S-Plus, es un lenguaje basado en S, pero con sus propias particularidades. La ventaja de esta herramienta respecto a S-Plus reside en su licencia de libre distribución. Por lo demás, tanto S-Plus como R son entornos de desarrollo similares, que comparten una interfaz similar de consola de comandos y una misma forma de interacción con el usuario. En concreto, el usuario maneja instrucciones que son una mezcla de llamadas a funciones y asignaciones. En este sentido, dado que ambas plataformas de desarrollo ya tienen implementadas de base gran cantidad de funciones estadísticas integradas en paquetes distintos, el usuario se limita normalmente a hacer uso de ellas a partir de sus especificaciones paramétricas.

Precisamente, este estilo de programación de alto nivel a partir de instrucciones interpretadas es en principio mucho más atractivo para el usuario no programador. De hecho, es un estilo de programación no viable en lenguajes compilados, que son la gran mayoría.

Sin embargo, aun a pesar de facilitar al usuario la ejecución de instrucciones sencillas y permitir a usuarios más avanzados la extensión de la funcionalidad del sistema, el trabajar con funciones es un estilo que recuerda más a la programación estructurada que no a la programación orientada a objetos. En concreto, el usuario llama a funciones que ejecutan una determinada operación y ofrecen un resultado inmediato en la misma consola de comandos. Este estilo no nos lleva a pensar en dichas funciones como métodos inherentes a un objeto y en los resultados de dichas funciones como datos que modifican el estado de ese objeto. Sin embargo, la realidad es que tanto S-Plus como R están basados en objetos internamente, aunque no permitan de forma directa el uso de referencias a dichos objetos (son lenguajes *basados en objetos*). El uso de referencias a objetos es el estilo de programación asociado a un sistema claramente *orientado a objetos*, como es el caso del lenguaje Java. Esa referencia a objetos es la que nos permite de forma directa instar a un determinado objeto X a que ejecute alguno de sus métodos, y solicitar que nos “enseñe” los resultados de ejecución del método empleado (estado actual del objeto).

La plataforma de desarrollo estadístico Omegahat es totalmente orientada a objetos en este sentido, pues su lenguaje Omegahat tiene de base al lenguaje Java. De hecho, Omegahat

permite un acceso directo a la implementación de programas en lenguaje Java, aprovechando todo el potencial de este último. Sin embargo, por conveniencias de uso interactivo, Omegahat suplementa la capacidad de evaluar expresiones Java individuales con el acceso a funciones interpretadas al estilo de S-Plus o R, las cuales son más simples de definir, modificar y usar que los métodos generales de Java. En este sentido, se puede considerar el lenguaje Omegahat como un híbrido de los lenguajes Java, S y R (Temple, 2000). La invocación de grupos de expresiones Omegahat se vuelve de esta manera una tarea simple, aunque sin perder de vista que cada variable del sistema es un objeto Java con sus propios métodos.

Como se ha indicado en el párrafo anterior, el lenguaje Omegahat permite el acceso directo a código Java desde la consola de comandos, de manera que una clase escrita en Java puede ser utilizada en Omegahat. Ahora bien, la definición de esa clase es fija durante la sesión, y si se redefine es necesario cerrar y reabrir Omegahat. Aunque se espera que las definiciones de las clases que se van a emplear no cambien, esta plataforma ofrece la posibilidad de crear y redefinir clases de forma dinámica en la sesión interactiva de usuario (se denominan *clases de usuario*). Es una alternativa cómoda a la compilación de código Java (*bytecode*). En definitiva, el entorno Omegahat ofrece al usuario la posibilidad de programar en todo momento "pensando en objetos", ventaja ésta no disponible de forma natural en los entornos[2] S-Plus y R.

Por su parte, Java también puede acceder a objetos Omegahat. Para ello, el entorno de desarrollo proporciona mecanismos para encapsular funciones y expresiones implementadas en lenguaje Omegahat, con la finalidad de que puedan ser accesibles como instancias (objetos) de cualquier clase Java. Gracias a ello, cualquier componente estadístico diseñado con Omegahat puede ser integrado como un componente independiente de Java. De hecho, los objetos Omegahat son objetos Java *per se*. Como consecuencia, ese componente puede ser ejecutado en cualquier plataforma física que disponga de una máquina virtual de Java (JVM). No sería necesario, por tanto, disponer del

[2] Actualmente existe un paquete para ambas plataformas de desarrollo (denominado OOP) que permite integrar en la sesión interactiva el uso de referencias a objetos y definiciones de métodos al estilo de lenguajes como Java y C++.

programa intérprete Omegahat para acceder a la funcionalidad de dicho elemento (ventaja no disponible en las plataformas S-Plus y R). Además, como componente orientado a objetos puede ser integrado junto a otros para extender un sistema dado. En este sentido, si se tiene en cuenta la facilidad actual para distribuir todo tipo de información por la red, incluyendo componentes de software, y que además cualquier ordenador puede disponer de la herramienta Omegahat y de la JVM, por ser ambas de libre distribución, nos podemos hacer una idea sobre lo fructífera que podría ser una inmersión conjunta de usuarios y desarrolladores estadísticos en esta nueva plataforma de desarrollo.

Es más, ello no implica que el desarrollo de plataformas evolucionadas como R y S-Plus vaya a quedar estancado, pues no hay que perder de vista que Java, gracias a su diseño y a la extensa funcionalidad que posee en cuanto a integración de componentes provenientes de otros entornos, es capaz de acoger como propios los objetos derivados de cualquier plataforma; y con más motivo si tenemos en cuenta que la base de ambos entornos es el lenguaje S, el cual ha evolucionado de forma importante hacia el modelo de orientación a objetos. Además, en la otra dirección, el lenguaje específico de Omegahat facilita la exportación de sus propias funciones Omegahat a dichos entornos.

En este sentido, Temple (2000), como integrante del proyecto Omegahat, presenta la herramienta Omegahat como una plataforma que ofrece nuevas opciones para la computación estadística y que abre nuevas posibilidades en cuanto a la integración de funcionalidad procedente de otros entornos de desarrollo estadístico.

4. EL PROCESO UNIFICADO DE DESARROLLO DE SOFTWARE

Una vez expuestas las tres fases clave del modelado orientado a objetos (análisis, diseño e implementación), el objetivo de este apartado es profundizar en la conexión de estas tres actividades de una manera adecuada y operativizada. La relación entre estas tres etapas y la forma de operativizar dicha relación se puede adecuar al estándar conocido como *Proceso Unificado de Desarrollo de Software* (Jacobson et al., 2000), generalmente referenciado como *Proceso Unificado* (UP).

Un proceso define quién está haciendo qué, cuándo, y cómo alcanzar un determinado objetivo. En la ingeniería del software el objetivo es construir un producto software o mejorar uno existente. Un proceso efectivo proporciona normas para el desarrollo eficiente de software de calidad.

Los autores del UP crean un modelo de proceso enfocado desde la perspectiva de desarrollo de software en el campo industrial y empresarial. En ese sentido, el proceso involucra tanto a las personas (sus habilidades o roles) pertenecientes a los diferentes niveles de organización de las empresas, como a las tecnologías –lenguajes de programación, sistemas operativos, ordenadores, entornos de desarrollo, estructuras de red, etc.– disponibles en el momento en que se va a emplear el proceso, y a las herramientas de software que se utilizan para automatizar las actividades definidas en el proceso (por ejemplo, una herramienta CASE de modelado para UML).

El UP es el producto final de tres décadas de desarrollo y uso práctico en el que han estado involucrados otros productos de proceso anteriores y que ha recibido aportaciones de muchas otras fuentes (Jacobson et al., 2000). Es un proceso con gran influencia actualmente en la industria del desarrollo de software, gracias a que se encuentra adaptado a las dificultades que afrontan los desarrolladores para coordinar las múltiples cadenas de trabajo de un gran proyecto de software. Se trata de un método común, unificado, ante la necesidad de integrar las múltiples facetas del desarrollo. Proporciona una guía para ordenar las actividades de un equipo, es un proceso que dirige las tareas de cada desarrollador por separado y del equipo como un todo, especifica además los *artefactos* (piezas de información tangible) que deben desarrollarse y ofrece criterios para el control y la medición de los productos y actividades del proyecto.

El UP se define como un proceso dirigido por *casos de uso*, centrado en la arquitectura, iterativo e incremental. Está basado en componentes (i.e., el sistema software en construcción está formado por componentes software) y utiliza el nuevo estándar de modelado visual UML para describir los distintos modelos generados durante el desarrollo del proceso, es decir, para preparar todos los esquemas de un sistema software. En este proceso, se distinguen cuatro fases fundamentales, dentro de cada una de las cuales se plantean una serie de flujos de trabajo iterativos que permiten la evolución del proyecto de manera incremental (figura 6):

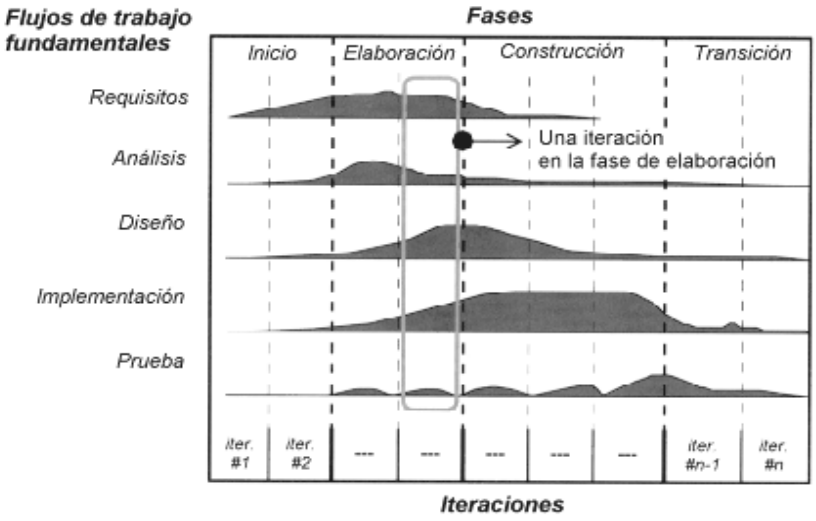


Figura 6. Fases y flujos de trabajo fundamentales en el ciclo de vida del desarrollo de software (Fuente: Jacobson et al., 2000)

Se diferencian cuatro fases en el proceso de desarrollo de software, atendiendo al momento en que se realizan: inicio, elaboración, construcción y transición. Cada una de estas fases se divide entonces en una o más iteraciones. En cada iteración, el proceso se detiene en mayor o menor grado en cada uno de los distintos flujos de trabajo. Se diferencian cinco flujos de trabajo[1] fundamentales en este proceso, atendiendo al estado de desarrollo del sistema: requisitos, análisis, diseño, implementación y prueba.

[1] Las fases de desarrollo descritas en el en el apartado anterior (análisis, diseño e implementación) aparecen integradas en este modelo de proceso como *flujos de trabajo* iterativos, proporcionando una visión adaptada del ciclo de vida orientado a objetos (figura 4, apartado 3).

En la fase de *inicio* se establece una visión del proyecto y su alcance, es cuando la idea inicial para el desarrollo se lleva al punto de estar suficientemente fundamentada para garantizar la entrada en la fase de elaboración. En esta fase de inicio se evalúa la viabilidad del proyecto, sobre todo cuando está en juego una gran inversión de recursos humanos y económicos. En este sentido, esta fase apenas consumirá dedicación cuando esos proyectos sean de pequeña escala. La captura de requisitos comienza en esta fase (*véase* figura 6), que como veremos en el apartado 4.1 es un flujo de trabajo que pretende modelar la funcionalidad del sistema acudiendo a un nivel de abstracción elevado. Para conseguir ese esquema de funcionalidad utiliza los llamados casos de uso. Un *caso de uso* es una secuencia de acciones que el sistema lleva a cabo para ofrecer algún resultado de valor para un *actor*. Un actor puede ser una persona humana, un dispositivo de hardware, u otro sistema. Los actores utilizan el sistema interactuando con los casos de uso.

La *elaboración* es la segunda fase del proceso, cuando se definen la visión del producto y su arquitectura. En esta fase se expresan con claridad los requisitos del sistema, proporcionando una arquitectura estable para guiar el sistema a lo largo del ciclo de vida. Esta arquitectura es la estructura central del sistema, la línea base, el armazón a partir del cual evolucionará el sistema hacia el producto final. Se dice que el UP está dirigido por los *casos de uso* (producto de la captura de requisitos), que otorgan esa funcionalidad necesaria para que el sistema evolucione. En consecuencia, la forma del sistema corresponde a la arquitectura y la función a los casos de uso. En esta misma fase, se lleva a cabo el análisis de los casos de uso capturados (*véase* apartado 4.2), una vista más detallada de la funcionalidad del sistema y que sirve como abstracción o simplificación del diseño del modelo (*véase* apartado 4.3), flujo de trabajo que se inicia también en esta fase. En la última iteración dentro de la fase de elaboración se observa (figura 6) cómo empieza a adquirir relevancia la implementación del diseño (*véase* apartado 4.4) asociado a los casos de uso considerados relevantes durante la captura de requisitos en dicha iteración. A su vez, los componentes obtenidos con la implementación del diseño son sometidos a un período de pruebas; en esta fase se prueba la línea base ejecutable de la arquitectura.

En cada iteración se identifican e implementan unos cuantos casos de uso. Cada iteración, excepto quizás la primera de todas de un proyecto, se dirige por los casos de uso a través de todos los flujos de trabajo, de los requisitos al diseño y a la prueba, añadiéndose un

incremento más en el desarrollo del sistema. Cada incremento del desarrollo es, por tanto, una realización funcional de un conjunto de casos de uso. En cada iteración, se toma otro conjunto de casos de uso para desarrollar, y se añaden a los de la iteración anterior.

En consecuencia, la arquitectura se desarrolla mediante iteraciones, principalmente durante la fase de elaboración. Cada iteración evoluciona comenzando con los requisitos y siguiendo con el análisis, diseño, implementación y pruebas, pero centrándose en los casos de uso relevantes desde el punto de vista de la arquitectura y de otros requisitos. El resultado al final de la fase de elaboración es una línea base de la arquitectura –un esqueleto del sistema con pocos "músculos" de software.

Durante la fase de *construcción* es cuando se desarrolla, también de forma iterativa e incremental, un producto completo que está preparado para la transición hacia la comunidad de usuarios. Esto significa describir los requisitos restantes, refinando el diseño y completando la implementación y las pruebas de software. En esta fase, por tanto, los distintos modelos del sistema –descritos en los siguientes subapartados de este bloque– van creciendo hasta completarse. La descripción de la arquitectura, sin embargo, no crece significativamente debido a que la mayor parte de esta arquitectura se definió durante la fase de elaboración.

Finalmente, durante la fase de transición, el software se despliega en la comunidad de usuarios. Una vez que el sistema ha sido puesto en manos de los usuarios finales, a menudo aparecen cuestiones que requieren un desarrollo adicional para ajustar el sistema, corregir algunos problemas no detectados o finalizar algunas características que habían sido pospuestas. Esta fase comienza normalmente con una versión beta del sistema, que luego será reemplazada por la versión definitiva del producto.

En el capítulo 2 se hizo referencia al modelado de la arquitectura de un sistema (apartado 2.2., figura 1), como forma de obtener una representación clara de los requisitos y diseño del sistema que se desea implementar. En el *Proceso Unificado*, esta arquitectura se captura en forma de cinco vistas que interactúan entre sí: una vista del modelo de casos de uso, una vista del modelo de análisis, una vista del modelo de diseño, una vista del modelo de despliegue y una vista del modelo de implementación. Cada una de estas vistas es una proyección del modelo correspondiente, en otras palabras, un extracto o parte de ese

modelo. En ese sentido, una vista del modelo de casos de uso, por ejemplo, se parece al propio modelo de casos de uso; tiene actores y casos de uso, pero solamente aquellos que son arquitectónicamente significativos, mientras que el modelo de casos de uso final contiene todos los casos de uso. Lo mismo ocurre con la vista de la arquitectura del modelo de diseño; es igual que un modelo de diseño, pero sólo representa el diseño de los casos de uso interesantes para la arquitectura.

El UP se centra en esta arquitectura, como se indica en su propia definición, y es utilizada como un artefacto básico para conceptualizar, construir, gestionar y hacer evolucionar el sistema en desarrollo. El papel de la arquitectura software es parecido al papel que juega la arquitectura en la construcción de edificios. El edificio se contempla desde varios puntos de vista: estructura, servicios, conducción de la calefacción, fontanería, electricidad, etc. Esto permite a un constructor ver una imagen completa antes de que comience la construcción. Análogamente, la arquitectura en un sistema software se describe mediante diferentes vistas del sistema en construcción. Cada vista es una proyección de la organización y estructura del sistema, centrada en un aspecto particular de ese sistema.

Por tanto, la arquitectura se expresa en forma de vistas de todos los modelos del sistema, los cuales juntos representan al sistema entero. Volviendo al símil “esqueleto-músculos” de un organismo, la arquitectura es análoga al esqueleto cubierto por la piel pero con muy poco músculo (el software) entre los huesos y la piel, sólo lo necesario para permitir que el esqueleto haga movimientos básicos. El sistema es el cuerpo entero con esqueleto, piel y músculos.

Otro aspecto principal del PU, también contemplado en su definición, es el papel que juegan los casos de uso en la dirección de este proceso de desarrollo (Jacobson et al., 1992). Los casos de uso enlazan los flujos de trabajo fundamentales (figura 7). El proyecto de desarrollo progresa a través de estos flujos de trabajo, que se inician a partir de los casos de uso. Las clases se recogen de las descripciones de los casos de uso a medida que las leen los desarrolladores, buscando clases que sean adecuadas para la realización de los casos de uso. Los casos de uso también ayudan a desarrollar interfaces de usuario que hagan más fácil a estos usuarios el desempeño de sus tareas.

Figura 7. Los casos de uso enlazan los flujos de trabajo fundamentales

Conseguir el equilibrio correcto entre los casos de uso y la arquitectura es algo muy parecido al equilibrado de la forma y la función en el desarrollo de cualquier producto. Se consigue con el tiempo, y la clave consiste precisamente en hacer uso del tercer aspecto principal del PU: la técnica de desarrollo iterativo e incremental. Esta técnica proporciona la estrategia para desarrollar un producto software en pasos pequeños manejables: planificar un poco; especificar, diseñar e implementar un poco; integrar, probar y ejecutar un poco el resultado de cada iteración. Si se está satisfecho con un paso, se continúa con el siguiente. En cada paso, se obtiene una retroalimentación que permite ajustar los objetivos para el siguiente paso. Cuando se han dado todos los pasos que se habían planificado, se dispone de un producto desarrollado al que pueden acceder los usuarios. En definitiva, este desarrollo iterativo e incremental permite la atenuación de posibles riesgos que impidan el buen fin del proyecto.

En los siguientes subapartados nos centramos en cada uno de los flujos de trabajo indicados (figura 7), detallando aquellos aspectos particulares que llevan a la construcción del modelo final correspondiente: modelo de casos de uso (establece los requisitos funcionales del sistema), modelo de análisis (establece un diseño de las ideas), modelo de diseño (establece el vocabulario del problema y su solución), modelo de implementación (establece las partes que se utilizarán para ensamblar y hacer disponible el sistema físico) y modelo de pruebas (establece las formas de validar y verificar el sistema). Aunque la descripción de cada uno de estos modelos se realice de forma separada en los sucesivos apartados, no hay que perder de vista que están totalmente relacionados y que el desarrollo de cada uno de ellos no se hace de una sola pasada en cada flujo de trabajo, sino que precisamente se lleva a cabo de manera iterativa e incremental. Por tanto, un sistema no es sólo la colección de sus modelos, contiene también las relaciones entre ellos; los modelos están estrechamente enlazados unos con otros mediante *trazas* (estereotipo de

dependencia). Estas trazas permiten conectar elementos de un modelo con elementos de otro (figura 8):

Figura 8. Relación entre modelos del sistema

Cada uno de estos modelos presenta un nivel de abstracción diferente. El nivel máximo de abstracción lo sustenta el modelo de casos de uso, aumentando el nivel de detalle en cada uno de los sucesivos modelos; en este sentido, el modelo de implementación es el que presenta el menor nivel de abstracción. Gracias a las relaciones de traza, los elementos de un modelo con un determinado nivel de abstracción pueden derivarse a partir de los elementos de otro modelo con un nivel de abstracción superior (menor detalle).

4.1. REQUISITOS DEL SISTEMA

Hay diferentes puntos de partida para la captura de requisitos del sistema. Cuando el software a desarrollar da soporte directamente al negocio de una empresa, se puede acudir al llamado *modelo de negocio*, como abstracción o esquema del modelo de casos de uso. El modelo de negocio permite comprender los procesos de negocio de la organización, en términos de casos de uso del negocio y actores del negocio; permite entender el contexto del que se van a extraer los casos de uso para el sistema software.

Cuando el software a desarrollar no tenga este soporte de negocio, se puede tener como entrada un modelo de objetos sencillo, como un *modelo de dominio*, el cual captura los tipos más importantes de objetos en el contexto del sistema (Larman, 2003). Los objetos (o clases) del dominio representan los elementos que existen o los eventos que suceden en el entorno en el que trabaja el sistema. Muchas de estas clases pueden obtenerse de una especificación de requisitos o mediante una entrevista con los expertos del dominio. El objetivo del modelado del dominio es establecer el contexto del sistema. Cuando este

dominio es pequeño, puede ser suficiente un glosario de términos para establecer dicho contexto, sin necesidad de recurrir al desarrollo de un modelo de objetos. Las clases del dominio y el glosario de términos se utilizan en el desarrollo de los modelos de casos de uso y de análisis. En concreto, pueden utilizarse para describir los casos de uso y diseñar la interfaz de usuario, y además, pueden sugerir clases internas al sistema en desarrollo durante el análisis (Jacobson et al., 2000).

El esfuerzo principal en la fase de requisitos es desarrollar un modelo del sistema que se va a construir, y la utilización de los casos de uso es una forma adecuada de crear ese modelo. Los casos de uso proporcionan un medio intuitivo y sistemático para capturar los requisitos funcionales del sistema. Mediante la utilización de los casos de uso, los analistas se ven obligados a pensar en términos de quiénes son los usuarios y qué necesidades u objetivos se deben cumplir. Además, los casos de uso dirigen todo el proceso de desarrollo, puesto que la mayoría de actividades como el análisis, diseño y prueba se llevan a cabo partiendo de los casos de uso. Esas dos razones, la captura de los requisitos del sistema y la dirección del proceso de desarrollo, son precisamente los motivos por los cuales los casos de uso se han hecho populares y se han adoptado universalmente (Jacobson y Christerson, 1995).

Un *caso de uso* es una secuencia de acciones que el sistema lleva a cabo para ofrecer algún resultado de valor para un *actor* (Jacobson et al., 2000). Normalmente, un sistema tiene muchos tipos de usuarios, y cada tipo de usuario se representa por un actor. Los actores utilizan el sistema interactuando con los casos de uso. De manera informal, los casos de uso son historias de uso de un sistema para alcanzar los objetivos marcados (Larman, 2003).

Los actores representan, por tanto, a los usuarios del sistema. Ayudan a definir el sistema y aportan una imagen clara de qué es lo que debería hacer el sistema. Por ejemplo, un actor puede suministrar una entrada al sistema y puede recibir información desde el sistema. Es importante observar que un actor interactúa con los casos de uso, pero no tiene control sobre ellos. Normalmente, un actor representa un rol que es jugado por una persona, un dispositivo de hardware o incluso otro sistema al interactuar con nuestro sistema.

Se elige un nombre para cada caso de uso de forma que haga pensar en la secuencia de acciones concreta que añade valor a un actor. En este sentido, el nombre de un caso de uso

a menudo comienza con un verbo, y debe reflejar cuál es el objetivo de la interacción entre el actor y el sistema. Por ejemplo, en el contexto del comercio electrónico, podemos hablar de los casos de uso *Consultar Catálogo* o *Efectuar Pedido*. El usuario que interactúa con estos casos de uso se representaría, por ejemplo, mediante el actor *Cliente On-Line*. La notación gráfica UML que identifica a un caso de uso y a un actor es la siguiente:

Un modelo de casos de uso es un modelo del sistema que contiene actores, casos de uso y sus relaciones. UML permite hacer uso de los *diagramas de casos de uso* (figura 9) para modelar la vista de casos de uso de un sistema. Una vista de casos de uso no es más que una proyección del modelo de casos de uso; en otras palabras, representa un extracto o parte del modelo, contiene actores y casos de uso arquitectónicamente significativos.

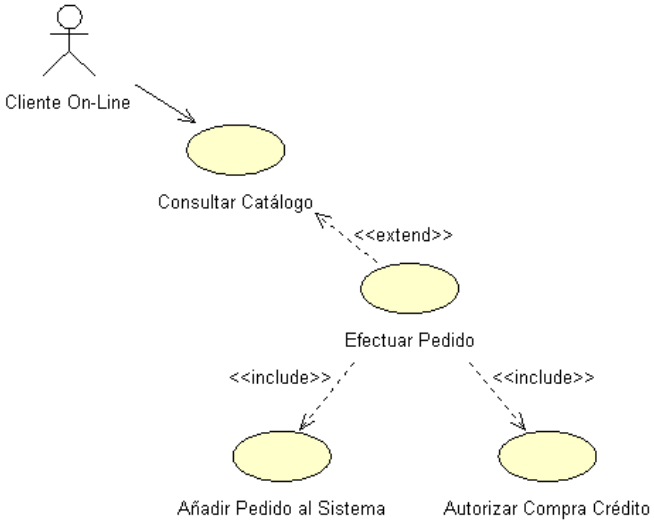


Figura 9. Diagrama de casos de uso (notación UML)

Este modelo de casos de uso evoluciona a lo largo de varios incrementos en el proceso de desarrollo, donde las iteraciones añadirán nuevos casos de uso y/o añadirán detalle a las descripciones de los casos de uso existentes.

Los casos de uso ayudan a los desarrolladores a encontrar las clases del sistema. En este sentido, las clases se pueden extraer de las descripciones de los casos de uso, descripciones que pueden comprender desde la simple redacción de una serie de frases que resumen las acciones, hasta la descripción paso a paso de lo que el sistema necesita hacer cuando interactúa con sus actores (Jacobson et al., 2000).

Algunas veces es difícil decidir el ámbito de un caso de uso. Una secuencia de interacciones usuario-sistema se puede especificar en un caso de uso o en varios, los cuales el actor invoca uno tras otro. Cuando se decide si un caso de uso candidato debe ser un caso de uso como tal, hay que considerar si es completo por sí mismo o si siempre se ejecuta como continuación de otro caso de uso (*include*: relación de inclusión, figura 9). Los casos de uso añaden valor a los actores. Para ser más específicos, un caso de uso entrega un resultado que se puede observar y que añade valor a un actor en concreto. Esta norma práctica para identificar un «buen» caso de uso puede ayudar a determinar el ámbito apropiado de uno de ellos.

Se ha hablado con anterioridad de las ventajas de los casos de uso en cuanto a su capacidad para dirigir el proceso de desarrollo. En este sentido, los casos de uso son un importante mecanismo para dar soporte a la trazabilidad a través de todos los modelos (figura 8). Un caso de uso en el modelo de requisitos es trazable a su realización en el análisis y en el diseño, a todas las clases participantes en su realización, a los componentes de software (indirectamente), y finalmente, a los casos de prueba que lo verifican. Esta trazabilidad es un aspecto importante de la gestión de un proyecto. Cuando se cambia un caso de uso, las realizaciones, clases, componentes y casos de prueba correspondientes tienen que comprobarse para ser actualizadas. De igual forma, cuando un componente de fichero (código fuente) se modifica, las clases, casos de uso y casos de prueba correspondientes que se ven afectados también deben comprobarse. La trazabilidad entre los casos de uso y el resto de elementos del modelo hacen más fácil mantener la integridad del sistema y conservar actualizado el sistema en su conjunto cuando tenemos requisitos cambiantes (Jacobson et al., 2000).

Se ha hecho referencia a los diagramas de casos de uso (figura 9) como una forma de describir la vista de casos de uso de un sistema, que contiene actores y casos de usos

significativos y sus relaciones. Cada actor juega un papel por cada caso de uso con el que colabora. Cada vez que un usuario en concreto (un humano u otro sistema) interactúa con el sistema, la instancia correspondiente del actor está desarrollando ese papel. Una instancia de un actor es por tanto un usuario concreto que interactúa con el sistema. Cualquier entidad que se ajuste a un actor puede actuar como una instancia del actor. Cada forma en que los actores usan el sistema se representa como un caso de uso. Los casos de uso son fragmentos de funcionalidad, especifican una secuencia de acciones que el sistema puede llevar a cabo interactuando con sus actores, incluyendo alternativas dentro de la secuencia. Por tanto, un caso de uso es una especificación; especifica el comportamiento de instancias de los casos de uso.

Por ejemplo, imaginemos la captura de un caso de uso denominado *Sacar Dinero*, que permite a las instancias del actor *Cliente del Banco* sacar dinero mediante un cajero automático. El caso de uso *Sacar Dinero* especifica las posibles instancias de ese caso de uso, es decir, las diferentes formas válidas de llevar a cabo el caso de uso por parte del sistema y la interacción requerida con las instancias de actores implicados. Supongamos que una persona concreta introduce en primer lugar su contraseña en el cajero, selecciona sacar 200 euros, y toma el dinero. El sistema habrá llevado a cabo una instancia del caso de uso. Si en cambio, esta persona introduce su contraseña, elige sacar 100 euros y después toma el dinero, el sistema habrá llevado a cabo otra instancia del caso de uso. Una tercera instancia del caso de uso podría ser lo que haría el sistema si esta persona solicita sacar 500 euros y el sistema no se lo permite debido a un saldo insuficiente o a una contraseña errónea.

Según el vocabulario de UML, un caso de uso es también un *clasificador* (mecanismo que describe características estructurales y de comportamiento), de forma que puede tener operaciones y atributos que se pueden representar igual que en las clases. Una descripción de un caso de uso puede por tanto incluir diagramas de estados, diagramas de actividades, diagramas de colaboración y diagramas de secuencia, en los que se emplean dichas operaciones y atributos para especificar el comportamiento del caso de uso. Los diagramas de estados especifican el ciclo de vida de las instancias de los casos de uso en términos de estados y transiciones entre los estados. Cada transición es una secuencia de acciones. Los diagramas de actividades describen el ciclo de vida con más detalle, especificando también

la secuencia temporal de acciones que tiene lugar dentro de cada transición. Los diagramas de colaboración y los de secuencia se emplean para describir las interacciones entre, por ejemplo, una instancia típica de un actor y una instancia típica de un caso de uso.

Una instancia de un caso de uso (*escenario* en terminología UML) es, por tanto, la realización (o ejecución) de un caso de uso. Otra forma de decirlo es que una instancia de un caso de uso es lo que el sistema lleva a cabo cuando "obedece a un caso de uso". Cuando se lleva a cabo una instancia de un caso de uso, ésta interactúa con instancias de actores, y ejecuta una secuencia de acciones según se especifica en el caso de uso. Esta secuencia se describe en un diagrama de estados o un diagrama de actividades; es un camino a lo largo del caso de uso. Puede haber distintos caminos, que son alternativas de la secuencia de acciones para el caso de uso. Veamos un ejemplo de diagrama de estados (notación UML) para un hipotético caso de uso denominado *Pagar Factura* (figura 10): muestra cómo una instancia del caso de uso *Pagar Factura* transita por diferentes estados (rectángulos redondeados) en una secuencia de transiciones de estado (flechas).

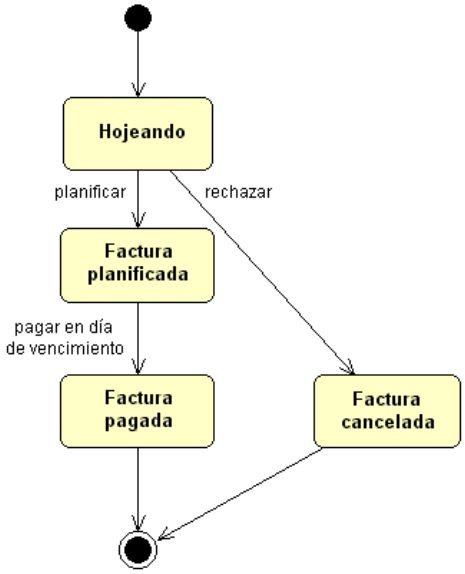


Figura 10. Diagrama de estados para el caso de uso *Pagar Factura*
(Fuente: Jacobson et al., 2000)

Las instancias de los casos de uso no interactúan con otras instancias de casos de uso. El único tipo de interacciones en el modelo de casos de uso tiene lugar entre instancias de

actores e instancias de casos de uso (Jacobson, 1995). Con ello, se consigue que el modelo de casos de uso sea simple e intuitivo, sin quedar atrapado en detalles. El comportamiento de cada caso de uso puede interpretarse, de esta manera, de manera independiente al de los otros casos de uso, lo cual hace más sencillo su modelado.

Es en el análisis y diseño del sistema (apartados 4.2 y 4.3, respectivamente) cuando aumenta el nivel de detalle de los casos de uso, manifestándose en forma de colaboraciones entre clases y/o subsistemas (agrupación de elementos). A este nivel de abstracción, se puede describir, por ejemplo, cómo una clase del modelo de análisis puede participar en varias realizaciones de casos de uso, aspecto éste que no se contempla en el modelo de casos de uso por cuestiones de simplicidad.

4.2. ANÁLISIS DE LOS REQUISITOS

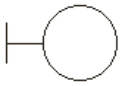
El análisis de los requisitos capturados en forma de casos de uso es un flujo de trabajo que permite de manera iterativa e incremental la construcción del modelo de análisis del sistema. Este modelo de análisis crece a medida que se analizan más y más casos de uso, ofreciendo una comprensión más precisa de los requisitos y una descripción de los mismos que sea fácil de mantener y que ayude a estructurar el sistema entero.

Precisamente, antes de comenzar a diseñar e implementar, debemos contar con una comprensión precisa y detallada de los requisitos. Llevando a cabo el análisis se consigue una separación de intereses que prepara y simplifica las subsiguientes actividades de diseño e implementación, delimitando los temas que deben resolverse y las decisiones que deben tomarse en esas actividades. En este sentido, el diseño y la implementación son mucho más que el análisis. En el diseño, se debe moldear el sistema y encontrar su forma: una forma que dé vida a todos los requisitos incorporados en el sistema; una forma que incluya componentes de código que se compilan e integran en versiones ejecutables del sistema durante la implementación; una forma que se mantenga firme bajo las presiones del tiempo, los cambios y la evolución; una forma con integridad (Jacobson et al., 2000).

Mediante esta separación de intereses, se puede afrontar el esfuerzo de desarrollo del software, y por tanto evitar la parálisis que puede ocurrir cuando se intentan resolver

demasiados problemas a la vez, incluyendo problemas que no hayan sido resueltos en absoluto debido a que los requisitos eran vagos y no se comprendían correctamente.

Uno de los más comunes grupos de elementos que encontramos en el análisis del modelo son las clases de análisis, a veces llamados objetos de análisis. Las *clases de análisis* son clases estereotipadas que representan un modelo conceptual para los elementos del sistema que tienen responsabilidad y comportamiento. Hay tres tipos de clases de análisis, las cuales son usadas en todo el modelo de análisis:

- Clases de interfaz: 
- Clases de entidad: 
- Clases de control: 

Una *clase de interfaz* es una clase estereotipada que modela la interacción entre uno o más actores (usuarios y sistemas externos) y el sistema. Esta interacción a menudo implica recibir (y presentar) información y peticiones de (y hacia) los usuarios y los sistemas externos. Las clases de interfaz modelan las partes del sistema que dependen de sus actores, lo cual implica que clarifican y reúnen los requisitos en los límites del sistema. Estas clases representan a menudo abstracciones de ventanas, formularios, paneles, interfaces de comunicaciones, interfaces de impresoras, sensores, terminales, etc.

Las clases de interfaz se mantienen en un nivel bastante alto y conceptual; es suficiente con que describan lo que se obtiene con la interacción (es decir, la información y las peticiones que se intercambian entre el sistema y sus actores). No es necesario que describan cómo se ejecuta físicamente la interacción, ya que esto se considerará en las actividades de diseño e implementación subsiguientes.

Cada clase de interfaz debería asociarse con al menos un actor, y viceversa.

Por ejemplo, la clase de interfaz *formulario de búsqueda* (figura 11), se utiliza para cubrir parte de la interacción entre el actor *Cliente On-Line* y el caso de uso *Consultar Catálogo* (interacción reflejada en un diagrama de casos de uso anterior, figura 9). Esta clase de

interfaz, junto a otras, se asocian con el “interior” del sistema, es decir, con las clases de control y de entidad.

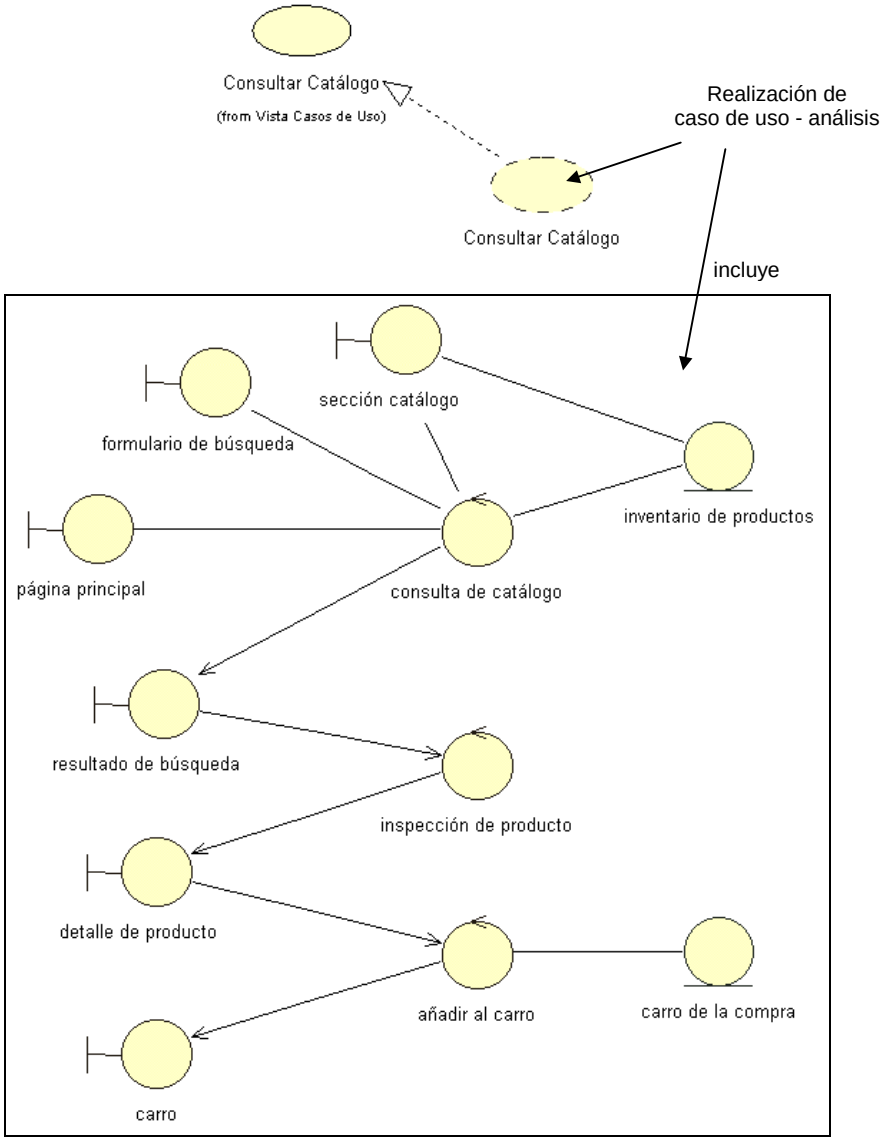


Figura 11. Un diagrama de clases de una realización[2] del caso de uso *Consultar Catálogo* (notación UML)

[2] Un caso de uso se define normalmente mediante una colaboración (elipse con línea discontinua), que representa la realización de dicho caso de uso. Un caso de uso se realiza creando una sociedad de clases y otros elementos que colaboran para llevar a cabo el comportamiento del caso de uso.

Las *clases de entidad*, por su parte, se utilizan para modelar información que posee una vida larga y que es a menudo persistente. Modelan la información y el comportamiento asociado de algún fenómeno o concepto, como una persona, un objeto del mundo real, o un suceso del mundo real. Un ejemplo de clase de entidad en la figura anterior es la clase denominada *inventario de productos* o la clase *carro de la compra*.

Las clases de entidad suelen mostrar una estructura de datos lógica y contribuyen a comprender de qué información depende el sistema. Además, poseen características persistentes que son frecuentemente reutilizadas en otros sistemas de casos de uso.

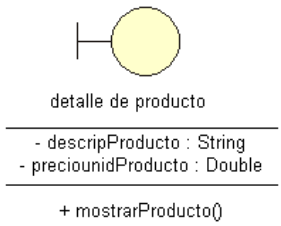
Las *clases de control*, por último, representan coordinación, secuencia, transacciones y control de otros objetos, y se usan con frecuencia para encapsular el control de un caso de uso en concreto. Un ejemplo de clase de control en la figura anterior es la clase denominada *añadir al carro*.

Los aspectos dinámicos del sistema se modelan con clases de control, debido a que ellas manejan y coordinan las acciones y los flujos de control principales, y delegan trabajo a otros objetos (es decir, objetos de interfaz y de entidad). Otro ejemplo de clase de control (figura 11) lo tenemos en la clase *inspección de producto*, que regula el proceso entre el resultado de la búsqueda presentado al usuario y el acceso de éste a los detalles de un determinado producto (información representada en las clases de interfaz *resultado de búsqueda* y *detalle de producto*, respectivamente).

Como se puede observar, la funcionalidad que definen los casos de uso se descompone de una forma orientada a objetos, en colaboraciones de objetos de análisis conceptuales. Esta descomposición proporciona una comprensión en profundidad de los requisitos.

Por tanto, el modelo de análisis es un modelo conceptual, es una especificación detallada de los requisitos y funciona como primera aproximación del modelo de diseño, siendo este último a su vez un esquema o abstracción de la implementación. Así, los desarrolladores utilizan este modelo de análisis para comprender de manera más precisa los casos de uso descritos en el flujo de trabajo de los requisitos, refinándolos en forma de colaboraciones entre clasificadores conceptuales (diferentes de los clasificadores de diseño que serán objeto de implementación). El término *clasificador*, como ya se comentó en el apartado anterior, indica que el elemento en cuestión (elemento *clase*, en este contexto) dispone de

atributos y operaciones propias. Se puede representar la clase *detalle de producto*, por ejemplo, con dicha información:



El diagrama de clases mostrado en la figura 11 se ha utilizado para describir la vista estática de una parte del sistema. De hecho, los diagramas de clases son los diagramas más comunes en el modelado de sistemas orientados a objetos. Para describir la vista dinámica de esa parte del sistema se puede acudir a los *diagramas de interacción*, que muestran una interacción que consta de un conjunto de objetos (instancias de las clases de análisis en este caso) y sus relaciones, incluyendo los mensajes que pueden enviarse entre ellos. Podemos distinguir dos tipos de diagramas de interacción: los *diagramas de secuencia* y los *diagramas de colaboración*. Cuando el objetivo es identificar secuencias de interacción detalladas, destacando la ordenación temporal de los mensajes, se utilizarán los diagramas de secuencia. Cuando el objetivo fundamental es identificar requisitos y responsabilidades sobre los objetos, destacando la organización estructural de los objetos que envían y reciben mensajes, se utilizarán los diagramas de colaboración (figura 12).

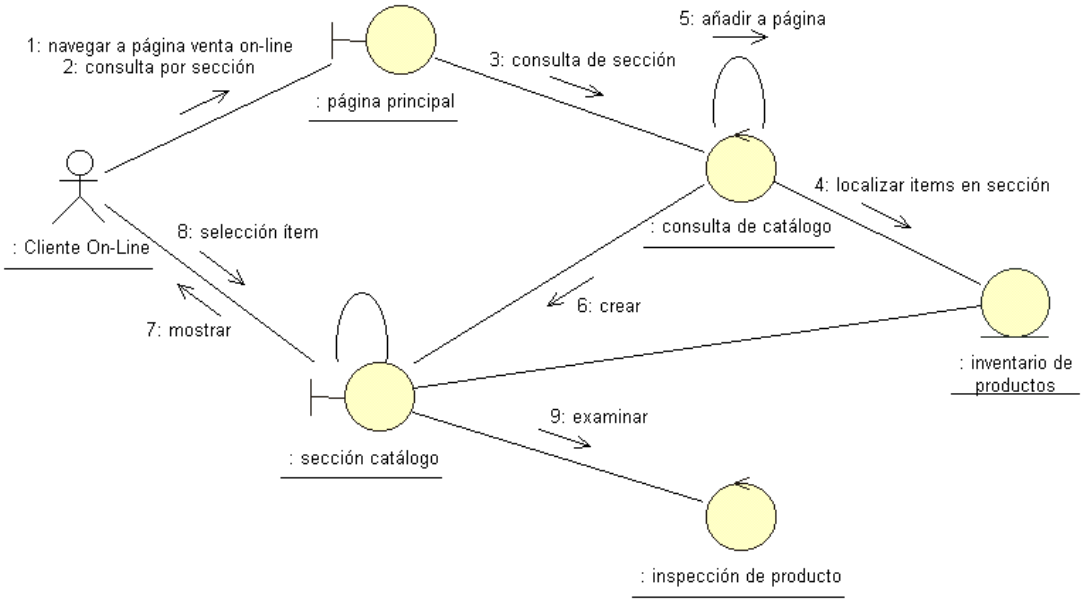


Figura 12. Un diagrama de colaboración asociado a la realización del caso de uso *Consultar Catálogo* (notación UML)

En este diagrama de colaboración se representa la interacción entre varias instancias de clases de análisis asociadas a la realización del caso de uso *Consultar Catálogo*. Concretamente, la colaboración entre estas clases representa una realización particular de dicho caso de uso en el contexto de la consulta por sección (información clasificada). Una consulta por búsqueda (uso de palabras clave) se llevaría a cabo empleando una dinámica de colaboración similar aunque sustituyendo, por ejemplo, la clase de interfaz *sección catálogo* por la clase de interfaz *formulario de búsqueda* (representada en el diagrama de clases, figura 11).

Hemos visto, por tanto, que dentro del modelo de análisis los casos de uso se describen mediante clases de análisis y sus objetos (instancias de las clases), aspecto que se refleja en diagramas estáticos y dinámicos asociados a la *realización de caso de uso – análisis*. Los aspectos dinámicos en relación a los cambios de estado de una instancia de una clase concreta pueden ser descritos mediante diagramas de estados y diagramas de actividades.

El modelo de análisis se representa mediante un sistema de análisis que denota el paquete de más alto nivel del modelo. La utilización de otros paquetes de análisis permite organizar el modelo en partes más manejables que representan abstracciones de subsistemas del

diseño del sistema. Un paquete de análisis puede constar de clases de análisis, de realizaciones de casos de uso (colaboraciones), y de otros paquetes de análisis. Los paquetes de análisis deberían incluir contenidos fuertemente relacionados, y a la vez, minimizar sus dependencias respecto a otros paquetes. Debido a que se capturan los requisitos funcionales del sistema en la forma de casos de uso, una forma directa de identificar paquetes del análisis es asignar cierto número de casos de uso implicados en un mismo proceso a un paquete concreto y después realizar la funcionalidad correspondiente dentro de ese paquete (Jacobson et al., 2000).

4.3. DISEÑO DE LA SOLUCIÓN

De igual forma que en el modelo de análisis, el modelo de diseño también define clasificadores –en este caso, se definen clases, subsistemas e interfaces–, relaciones entre estos clasificadores, y colaboraciones que llevan a cabo los casos de uso (las realizaciones de los casos de uso). Sin embargo, los elementos definidos en el modelo de análisis son más conceptuales, mientras que el modelo de diseño es más "físico" por naturaleza, sus elementos se adaptan al entorno de la implementación.

Puede hacerse la traza de una realización de caso de uso en el modelo de análisis a partir de una realización de caso de uso en el modelo de diseño (figura 13):

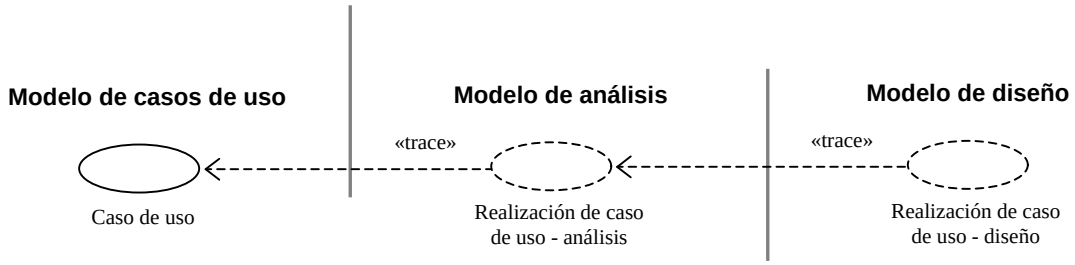


Figura 13. Realizaciones de caso de uso en diferentes modelos

Las realizaciones de caso de uso en los diferentes modelos sirven para propósitos distintos. Cuando se definen clases de análisis la intención es proporcionar una comprensión precisa y detallada de los requisitos, mediante un modelo de análisis genérico respecto al diseño (aplicable a varios diseños); son clases independientes de la implementación. En cambio,

cuando se diseñan esas clases de análisis, todas ellas especifican y hacer surgir clases de diseño más refinadas que se adaptan al entorno de implementación; se obtiene un modelo de diseño no genérico, específico para una implementación (basada en un lenguaje de programación específico).

Por tanto, una entrada esencial en el diseño es el resultado del análisis, que impone una estructura que interesa conservar lo más fielmente posible cuando se de forma al sistema. En concreto, los propósitos del diseño son (Jacobson et al., 2000):

- Adquirir una comprensión en profundidad de los aspectos relacionados con los requisitos no funcionales y restricciones relacionadas con los lenguajes de programación, componentes reutilizables, sistemas operativos, tecnologías de distribución y concurrencia, tecnologías de interfaz de usuario, tecnologías de gestión de transacciones, etc.
- Crear una entrada apropiada y un punto de partida para actividades de implementación subsiguientes capturando los requisitos o subsistemas individuales, interfaces y clases.
- Ser capaces de descomponer los trabajos de implementación en partes más manejables que puedan ser llevadas a cabo por diferentes equipos de desarrollo, teniendo en cuenta la posible concurrencia. Esto resulta útil en los casos en los que la descomposición no puede ser hecha basándose en los resultados de la captura de requisitos (incluyendo el modelo de casos de uso) o análisis (incluyendo el modelo de análisis). Un ejemplo podría ser aquellos casos en los que la implementación de estos resultados no es directa.
- Capturar las interfaces entre subsistemas, considerando las dependencias entre estos subsistemas. Las interfaces proporcionadas por un subsistema definen operaciones que son accesibles desde fuera del mismo.
- Ser capaces de visualizar y reflexionar sobre el diseño utilizando una notación común.
- Crear una abstracción de la implementación del sistema, en el sentido de que la implementación es un refinamiento directo del diseño, que rellena lo existente sin

cambiar la estructura. Esto permite la utilización de tecnologías como la generación de código y la ingeniería de ida y vuelta entre el diseño y la implementación.

Mientras que el modelo de análisis representa el sistema en términos de objetos que expertos en el dominio conocen, el modelo de diseño representa el mismo sistema pero con un nivel de abstracción cercano al código fuente.

El diseño es el centro de atención al final de la fase de elaboración y el comienzo de las iteraciones de construcción. Esto contribuye a una arquitectura estable y sólida y a crear un plano del modelo de implementación. Más tarde, durante la fase de construcción, cuando la arquitectura es estable y los requisitos están bien entendidos, el centro de atención se desplaza a la implementación (implementación del diseño). No obstante, el modelo de diseño está muy cercano al de implementación. Esto es especialmente cierto en la ingeniería de ida y vuelta, donde el modelo de diseño se puede utilizar para visualizar la implementación y para soportar las técnicas de programación gráfica. La *ingeniería de ida y vuelta* es un concepto que representa la combinación de la *ingeniería directa* y la *ingeniería inversa*. La ingeniería directa permite la transformación de un modelo en código a través de su traducción a un determinado lenguaje de implementación, mientras que la ingeniería inversa transforma el código en un modelo a través de su traducción desde un determinado lenguaje de implementación.

El modelo de diseño es un modelo de objetos que describe la relación física de los casos de uso centrándose en cómo los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar. Además, el modelo de diseño sirve de abstracción de la implementación del sistema y es, de ese modo, utilizada como una entrada fundamental de las actividades de implementación.

El modelo de diseño se representa por un sistema de diseño que denota el subsistema de nivel más alto del modelo. La utilización de otros subsistemas es una forma de organización del modelo de diseño en porciones más manejables. En este sentido, el modelo de diseño se puede ver como una jerarquía de subsistemas de diseño que contienen clases de diseño, realizaciones de caso de uso-diseño e interfaces (las interfaces se utilizan para especificar las operaciones que proporcionan las clases y los subsistemas del diseño).

Los casos de uso son realizados por las clases de diseño y sus objetos en el modelo de diseño. Esto se representa por colaboraciones en el modelo de diseño, que denotan la realización de caso de uso – diseño. La *realización de caso de uso – diseño* es diferente de la *realización de casos de uso – análisis* (*véase* figura 13). Esta última describe cómo se realiza un caso de uso en términos de interacción entre objetos del análisis, mientras que la primera representa cómo se realiza un caso de uso en términos de interacción entre objetos del diseño.

El diseño de una clase incluye el mantenimiento del diseño en sí mismo y los siguientes aspectos de éste (Jacobson et al., 2000):

- Sus operaciones.
- Sus atributos
- Las relaciones en las que participa.
- Sus métodos (que realizan sus operaciones).
- Los estados impuestos.
- Sus dependencias con cualquier mecanismo de diseño genérico.
- Los requisitos relevantes a su implementación.
- La correcta realización de cualquier interfaz requerida.

Las operaciones que las clases de diseño van a necesitar se describen utilizando la sintaxis de los lenguajes de programación. Esto incluye especificar la visibilidad de cada operación (por ejemplo, *public*, *protected*, *private* en Java o C++). Las operaciones de las clases de diseño necesitan soportar todos los roles que las clases desempeñan en las diferentes realizaciones de casos de uso. En otras palabras, la clase debe implementar aquellas operaciones que pueda emplear cualquiera de las realizaciones de casos de uso en la que participa. Por ejemplo, la clase *Factura* puede participar en la realización de los casos de uso *Pagar Factura*, *Enviar Aviso* y *Enviar Factura al Comprador*. Cada una de estas realizaciones de casos de uso utiliza objetos *Factura* de forma diferente, leyendo y/o cambiando el estado de dichos objetos a su manera; para ello, acuden a operaciones adecuadas implementadas en la clase *Factura*, por ejemplo, el uso de los métodos *crear( )*, *enviar( )*, *planificar( )* y *cerrar( )*. Así, el caso de uso *Enviar Factura al Comprador* crea y

envía facturas, el caso de uso *Pagar Factura* planifica facturas, y así cada una de las realizaciones de caso de uso que empleen dicha clase.

Las operaciones utilizadas por los objetos de una clase, de hecho, podrían estar recogidas en una interfaz o en una clase abstracta que declare métodos virtuales (código vacío). Las clases que accedan a dichos métodos (vía herencia o vía vínculo a interfaz), desarrollarán una implementación particular de los mismos. Como se dijo en su momento, este planteamiento favorece la extensión de la funcionalidad del sistema.

Los atributos asociados a las clases de diseño también se describen utilizando la sintaxis del lenguaje de programación. Un atributo especifica una propiedad de una clase de diseño y está a menudo implicado y es requerido por las operaciones de la clase.

Al igual que el modelo de análisis, el modelo de diseño puede ser descrito gráficamente mediante diagramas de clases (vista estática) y mediante diagramas de interacción (vista dinámica). Ejemplos de diagramas de clases de diseño fueron mostrados en su momento (figura 3, apartado 2.3.5; figura 5, apartado 3.2). Además, mediante el uso de diagramas de estados y diagramas de actividades se puede modelar el flujo de un objeto o instancia de una clase concreta conforme cambia de estado a partir de la ejecución de determinadas operaciones asociadas a la clase. Veamos, por ejemplo, a partir de un diagrama de estados, cómo cambia de estado un objeto concreto de la clase *Factura* (figura 14).

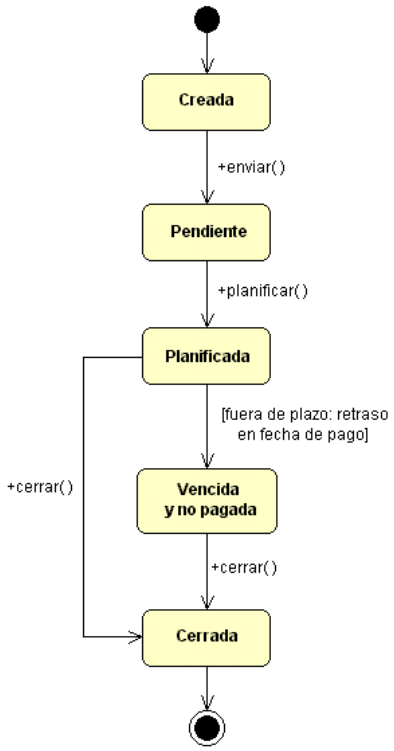


Figura 14. Un diagrama de estados para la clase *Factura*
(Fuente: Jacobson et al., 2000)

Las relaciones en las que puede participar una clase se clasifican básicamente en cuatro tipos importantes: dependencias, generalizaciones, realizaciones y asociaciones (Booch et al., 1999). Las *dependencias* indican que un elemento (una clase, en este caso) utiliza a otro. Las *generalizaciones* (herencia) conectan clases generales con otras más especializadas en lo que se conoce como relaciones subclase/superclase. Las *realizaciones*, en este contexto, especifican la relación entre una interfaz y una clase. Una interfaz es una colección de operaciones que sirven para declarar un servicio de una clase o de un componente, como un contrato que debe llevar a cabo (realizar) esa clase o componente. Las *asociaciones* son relaciones estructurales entre instancias, que especifican que los objetos de un elemento están conectados con los objetos de otros. Dada una asociación simple entre dos clases, se puede navegar (transmitir mensajes) desde un objeto de una clase hasta un objeto de la otra clase, y viceversa. En el apartado 5, dedicado a UML, se detalla la notación gráfica de cada tipo de relación, con sus variantes.

Podemos esbozar inicialmente algunas clases de diseño a partir de las clases significativas para la arquitectura del modelo de análisis. Además, se pueden utilizar las relaciones entre estas clases de análisis para identificar un conjunto tentativo de relaciones entre las correspondientes clases de diseño. En proyectos de pequeña magnitud, posiblemente la correspondencia entre la estructura de análisis y la estructura de diseño sea prácticamente directa, mediante la constitución de trazas de una sola clase de diseño a la clase de análisis correspondiente, y la adaptación de los atributos y operaciones de dicha clase de análisis a un entorno de implementación específico.

Por otro lado, si el nivel de abstracción asumido en el modelo de análisis es muy alto (como ocurrirá seguramente en proyectos de gran magnitud), probablemente se derivarán varias clases de diseño a partir de una sola clase de análisis.

En consecuencia, el abordar proyectos de pequeña magnitud, nos lleva incluso a plantear la posibilidad de establecer una traza del modelo de diseño directamente al modelo de casos de uso. Cuando el número de elementos del sistema es reducido, puede que un análisis detallado de los mismos en un modelo de análisis sea innecesario; sobre todo si se tiene en cuenta que el modelo de casos de uso del sistema ya integra un análisis del contexto del problema, y que en esta situación, probablemente el modelo de análisis no contribuya de una manera significativa al desarrollo del diseño de la solución. Por tanto, en este caso quizás sea más cómodo para el desarrollador conectar el modelo de casos de uso directamente con el modelo de diseño.

De hecho, dentro del *Proceso Unificado*, el modelo de análisis es considerado un modelo opcional, al igual que el modelo de procesos (Booch et al., 1999, Apéndice C). En este sentido, Booch et al. (1999, p. 86) afirman que si se modela una simple aplicación que se ejecuta en una única máquina, se podría necesitar sólo el siguiente grupo de diagramas (en cada una de las vistas de la figura 1, apartado 2.2):

- Vista de casos de uso: Diagramas de casos de uso.
- Vista de diseño: Diagramas de clases (para modelado estructural). Diagramas de interacción (para modelado del comportamiento).

- Vista de procesos: No se requiere.
- Vista de implementación: No se requiere.
- Vista de despliegue: No se requiere.

La vista de procesos de un sistema comprende los hilos y procesos que forman los mecanismos de sincronización y concurrencia del sistema. Esta vista cubre principalmente el funcionamiento, capacidad de crecimiento y rendimiento del sistema. Con UML, los aspectos estáticos y dinámicos de esta vista se capturan en el mismo tipo de diagramas que la vista de diseño, pero con énfasis en las clases activas que encarnan estos hilos y procesos. Las clases activas representan procesos que organizan el trabajo de las otras clases (no activas). Una clase activa es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución y por lo tanto pueden dar origen a actividades de control. Un *proceso* se define como un flujo de control pesado que puede ejecutarse concurrentemente con otros procesos. Un *hilo* (*thread*), en cambio, es un flujo de control ligero que se puede ejecutar concurrentemente con otros hilos en el mismo proceso. Por tanto, una clase activa es igual que una clase, excepto en que sus objetos representan elementos cuyo comportamiento es concurrente con otros elementos.

La vista de implementación, como proyección del modelo de implementación (*véase* siguiente subapartado), comprende los componentes y archivos que se utilizan para ensamblar y hacer disponible el sistema físico. Con UML, los aspectos estáticos de esta vista se capturan en los diagramas de componentes; los aspectos dinámicos se capturan en los diagramas de interacción, diagramas de estados y diagramas de actividades.

La vista de despliegue contiene los nodos que forman la topología hardware sobre la que se ejecuta el sistema. Esta vista se preocupa principalmente de la distribución, entrega e instalación de las partes que constituyen el sistema físico. Cada nodo representa un recurso de computación, normalmente un procesador o un dispositivo hardware similar. Además, los nodos poseen relaciones que representan medios de comunicación entre ellos, tales como *Internet*, *intranet*, *bus*, y similares. Con UML, los aspectos estáticos de esta vista se capturan en los diagramas de despliegue; los aspectos dinámicos de esta vista se capturan en los diagramas de interacción, diagramas de estados y diagramas de actividades.

4.4. IMPLEMENTACIÓN DEL DISEÑO

El modelo de implementación de un sistema describe cómo se implementan los elementos del modelo de diseño en términos de componentes como archivos de código fuente, ficheros de código binario, ejecutables y similares.

Cada clase de un subsistema de diseño debe ser implementada mediante componentes en el subsistema de implementación correspondiente. Los subsistemas de implementación proporcionan una forma de organizar los elementos del modelo de implementación en trozos más manejables. Un subsistema puede estar formado por componentes, interfaces y otros subsistemas. Además, un subsistema puede implementar (y así proporcionar) las interfaces que representan la funcionalidad que exportan en forma de operaciones.

Es importante entender que un subsistema de implementación se manifiesta a través de un «mecanismo de empaquetamiento» concreto en un entorno de implementación específico (por ejemplo, un *paquete* en Java).

Los subsistemas de implementación están muy relacionados con los subsistemas de diseño en el modelo de diseño. De hecho, los subsistemas de implementación deberían seguir la traza uno a uno (de manera isomórfica) de sus subsistemas de diseño correspondientes.

Los componentes y los subsistemas de implementación pueden tener «dependencias de uso» sobre interfaces. Un componente que realiza (y, por tanto, proporciona para su uso) una interfaz ha de implementar correctamente todas las operaciones definidas por dicha interfaz. Un componente que representa una clase de diseño puede realizar interfaces si lo permite el lenguaje de programación. Por ejemplo, una clase de diseño implementada en Java puede llevar a efecto esta realización, comprometiéndose a implementar cada una de las operaciones de la interfaz. Un subsistema de implementación que proporciona una interfaz tiene que contener los componentes u otros subsistemas que proporcionen dicha interfaz.

A partir de los *diagramas de componentes* se describen los aspectos estáticos de un sistema físico (*véase* figura 15). En este tipo de diagramas pueden aparecer, entre otros, ficheros

que representan código fuente (ficheros '.java' en Java o ficheros '.cc' en C++, por ejemplo), ficheros compilados que representan código objeto (por ejemplo, ficheros '.class' en Java o ficheros '.dll' en entorno Windows), y ficheros directamente ejecutables (ficheros '.exe' en entorno Windows, por ejemplo).

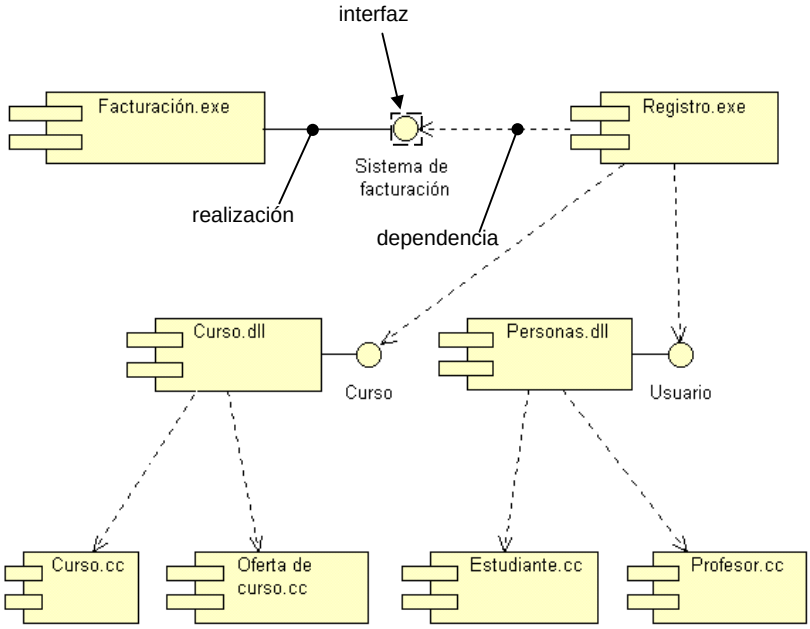


Figura 15. Un diagrama de componentes (notación UML)

En este diagrama (figura 15) aparecen componentes que realizan interfaces (por ejemplo, el fichero C*urso.dll* realiza la interfaz *Curso*), junto a otros componentes que acceden (relación de dependencia) a los servicios implementados (por ejemplo, el fichero *Registro.exe* accede a los servicios de interfaz proporcionados por el fichero *Curso.dll*).

Los componentes ejecutables del sistema se identifican a partir de las clases activas. Para determinar estos componentes ejecutables consideramos las clases activas encontradas durante el diseño y asignamos un componente ejecutable por cada clase activa, denotando así un proceso. Un *proceso* no es más que un programa ejecutándose de forma independiente y con un espacio propio de memoria; un sistema operativo multiproceso es capaz de ejecutar más de un proceso simultáneamente. Por otra lado, un *hilo* es un flujo secuencial simple dentro de un proceso. En este sentido, un proceso puede tener varios hilos ejecutándose. Por ejemplo, el programa *Internet Explorer* sería un proceso, mientras

que cada una de las ventanas que se pueden tener abiertas simultáneamente para acceder a páginas Web estaría formada por al menos un hilo.

Los objetos de una clase activa mantienen su propio hilo de control y se ejecutan concurrentemente con otros objetos activos[3]. No obstante, las clases de diseño no están normalmente activas, lo que implica que sus objetos se ejecutan en el espacio de direcciones y bajo el control de otros objetos activos. La semántica detallada de implementación depende del lenguaje de programación y de las tecnologías de distribución y concurrencia que se utilicen.

El propósito de la implementación de una clase es implementar una clase de diseño en un componente fichero. Esto incluye lo siguiente:

- Esbozo de un componente fichero que contendrá el código fuente.
- Generación de código fuente a partir de la clase de diseño y de las relaciones en que participa.
- Implementación de las operaciones de la clase de diseño en forma de métodos.
- Comprobación de que el componente proporciona las mismas interfaces que la clase de diseño.

El código fuente que implementa una clase de diseño reside en un componente fichero. Por tanto, se ha de esbozar el componente fichero y considerar su ámbito. Es normal implementar varias clases de diseño en un mismo fichero. Se ha de decir, sin embargo, que el tipo de modularización y las convenciones de los lenguajes de programación en uso restringirán la forma en que los componentes de fichero son esbozados. Por ejemplo, cuando se usa Java, se crea un componente fichero ".java" para cada clase de implementación. En general, los componentes fichero elegidos deberían facilitar la compilación, instalación y mantenimiento del sistema (Jacobson et al., 2000).

[3] Realmente, esa concurrencia es sólo aparente, puesto que normalmente las plataformas tienen una sola CPU. Es el sistema operativo el que se encarga de generar la ilusión de que todo se ejecuta a la vez. En plataformas con varias CPU, sí que es posible que los procesos se ejecuten realmente a la vez, dedicándose cada CPU a un proceso distinto.

Durante el diseño, muchos de los detalles relacionados con la clase de diseño y con sus relaciones son descritos utilizando la sintaxis del lenguaje de programación elegido, lo que permite que la generación de partes del código fuente que implementan la clase sea más fácil. En particular, esto es así para las operaciones y atributos de la clase, y para las relaciones en las que la clase participa. En dichas operaciones, el nivel de detalle se refiere normalmente a su signatura (nombre del método, número y tipo de argumentos), puesto que la definición de su código se realiza en la fase de implementación.

La implementación de una operación incluye la elección de un algoritmo y unas estructuras de datos apropiadas, y la codificación de las acciones requeridas por el algoritmo. Los estados descritos para la clase de diseño pueden influir en el modo en que son implementadas las operaciones, puesto que estos estados determinan su comportamiento cuando recibe un mensaje (*véase* figura 14, apartado 4.3).

4.5. MODELO DE PRUEBAS

En la etapa de prueba, cada construcción (componente ejecutable) que se ha generado durante la implementación es la entrada que se somete a pruebas de integración y a pruebas de sistema. Las pruebas de integración son necesarias para cada construcción dentro de la iteración, mientras que las pruebas de sistema son necesarias sólo al final de la iteración (Jacobson et al., 2000). El modelo de pruebas precisamente describe cómo los componentes ejecutables del modelo de implementación son probados.

Durante la prueba, se verifica que el sistema implementa correctamente su especificación. Se desarrolla un modelo de pruebas compuesto por *casos de prueba* (especifican qué probar, con qué entradas y resultados y bajo qué condiciones) y *procedimientos de prueba* (especificación de cómo llevar a cabo uno o varios casos de prueba o parte de ellos). Se ejecutan los casos de prueba para estar seguros de que el sistema funciona como se esperaba.

El *Proceso Unificado* comenzaba con la captura de los casos de uso, pasando después a analizar, diseñar e implementar un sistema que llevaba a cabo esos casos de uso. Ahora, se describe cómo probar que los casos de uso se han implementado correctamente. De hecho,

mediante la identificación temprana de los casos de uso, se puede comenzar pronto la planificación de las actividades de prueba, y se pueden proponer casos de prueba desde el comienzo. Estos casos de prueba podrán detallarse mejor durante el diseño, cuando se sepa más sobre cómo el sistema realizará los casos de uso.

Las pruebas de los casos de uso pueden llevarse a cabo de dos formas distintas (figura 16): bien desde la perspectiva de un actor que considera el sistema como una caja negra (el caso de prueba sigue la traza de un caso de uso en el modelo de casos de uso), o bien desde una perspectiva de diseño (prueba de caja blanca), en la que el caso de prueba se construye para verificar que las instancias de las clases participantes en la realización del caso de uso hacen lo que deberían hacer (el caso de prueba deriva de una realización de caso de uso en el modelo de diseño). Las pruebas de caja negra pueden identificarse, especificarse y planificarse tan pronto como los requisitos sean algo estables; este tipo de pruebas evalúan el comportamiento externo observable del sistema. Por su parte, las pruebas de «caja blanca» evalúan la interacción entre los componentes que implementan un determinado caso de uso.

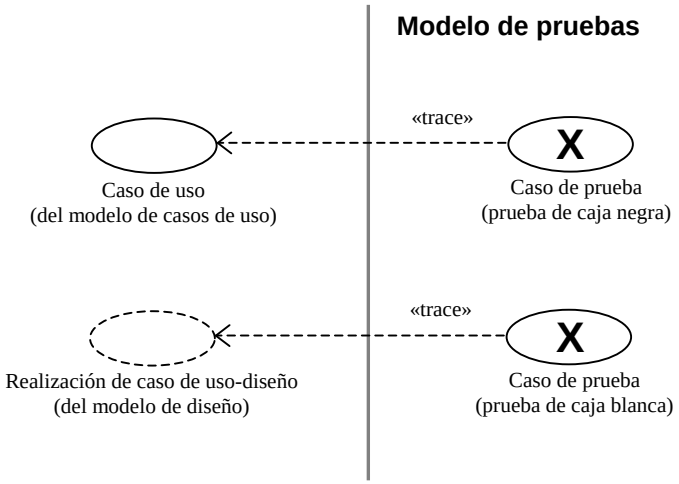


Figura 16. Dos formas distintas de realizar un caso de prueba
(Fuente: Jacobson et al., 2000)

Llegados a este punto, planteamos una breve discusión sobre la aplicación del *Proceso Unificado* (UP) en el desarrollo del bloque empírico de este trabajo (parte II). Ha quedado patente que este estándar de desarrollo de software tiene su máxima razón de ser en el contexto de proyectos de peso dentro de las empresas, proyectos que cubren necesidades que requieren automatizar tareas de gestión internas a la empresa y tareas de interacción con el exterior. Normalmente, estos proyectos tienen una larga duración y se invierte una gran cantidad de recursos para llevarlos a buen fin. Una forma de asegurar la calidad del producto, y sobre todo, de conseguir la adaptación del producto a la evolución de los sistemas, precisamente pasa por integrar dichos proyectos en un estándar de desarrollo (UP), que como tal, reduce riesgos durante todo el ciclo de vida del desarrollo del producto y sus posteriores actualizaciones.

Como se expuso al principio de este capítulo, este estándar define quién está haciendo qué a lo largo del ciclo de desarrollo de software. Se utiliza el término *trabajador* para denominar el puesto que puede ser asignado a una persona o equipo, y que requiere responsabilidades y habilidades concretas. Un *tipo de trabajador* es un papel que un individuo puede desempeñar durante el desarrollo del software; puede ser un analista del sistema, un especificador de casos de uso, un arquitecto, un ingeniero de casos de uso, un ingeniero de componentes, un integrador de sistemas, un diseñador de pruebas, un ingeniero de pruebas de integración o un ingeniero de pruebas del sistema (Jacobson et al., 2000). De hecho, una persona puede ser muchos trabajadores durante la vida de un proyecto, puede comenzar como especificador de casos de uso, y después pasar a ser ingeniero de componentes.

Es importante aplicar el UP al desarrollo de software estadístico, sobre todo si se plantean proyectos de larga duración que involucren a diferentes personas especialistas en un área de trabajo específico. Para ello, se debería establecer un sistema de distribución de tareas que redundara en una eficaz y eficiente aplicación del proceso. Por otro lado, si se está interesado en la creación de una herramienta de software específica, no excesivamente compleja, la flexibilidad en la adopción de estas pautas es un elemento a tener en cuenta; de hecho, ya se ha indicado que algunos modelos del proceso pueden ser opcionales en el desarrollo de una aplicación concreta.

En este sentido, en la segunda parte de este trabajo se desarrolla el análisis y diseño orientado a objetos de un *framework*[4] para el modelado estadístico con MLG, con la intención de ejemplificar un uso flexible del UP y demostrar que los artefactos derivados de este proceso (publicados bajo especificación UML) permiten acceder a una compresión precisa de los requisitos del sistema y del diseño de la solución.

[4] Un *framework* representa un patrón arquitectónico que proporciona una plantilla extensible para aplicaciones dentro de un dominio (Booch et al., 1999).

5. EL LENGUAJE UNIFICADO DE MODELADO

El *Lenguaje Unificado de Modelado*, o UML (*Unified Modeling Language*), es un lenguaje gráfico de modelado que provee de una sintaxis para describir los elementos importantes (llamadas *artefactos* en UML) de un sistema de software. UML, en cuanto a notación de modelado, es el sucesor de la oleada de métodos de análisis y diseño orientados a objetos que aparecieron a principios de los 90. De hecho, como se dijo en su momento, unifica la notación de los métodos de Booch, Rumbaugh (OMT) y Jacobson (OOSE), y además, es considerado un estándar OMG (OMG, 2003) (*véase* apartado 2.3.3).

Sin embargo, UML no es un método. Un método consiste, en principio, en un lenguaje de modelado y un proceso. UML aporta la parte de lenguaje de modelado, pero es necesario un proceso que utilice dicho lenguaje. Como se recoge en el capítulo anterior, el *Proceso Unificado de Desarrollo de Software*, creado por los mismos autores de UML, aporta la parte de proceso en el que se emplea este lenguaje de modelado.

Si bien ya se ha hablado en anteriores apartados del lenguaje UML y empleado para describir determinados ejemplos extraídos del contexto del proceso de desarrollo de software, en este capítulo nos centramos en aquellos aspectos del lenguaje considerados relevantes para la mayoría de problemas a modelar durante dicho proceso y que no han sido detallados suficientemente con anterioridad. En este sentido, Booch et al. (1999) afirman que el 80 por 100 de la mayoría de problemas puede modelarse con el 20 por 100 de UML. Los elementos estructurales básicos, tales como clases, atributos, operaciones, casos de uso, componentes y paquetes, junto a las relaciones estructurales básicas, tales como dependencia, generalización y asociación, son suficientes para crear modelos estáticos para muchos tipos de dominios de problemas. Si a esta lista se añaden los elementos de comportamientos básicos, tales como las máquinas de estados simples y las interacciones, se pueden modelar muchos aspectos útiles de la dinámica de un sistema. Sólo hará falta utilizar las características más avanzadas de UML cuando se empiece a modelar aspectos relacionadas con situaciones más complejas, como cuando se debe tratar con mecanismos de concurrencia y de distribución.

El vocabulario de UML incluye tres grandes bloques de construcción (Booch et al., 1999; OMG, 2003): elementos, relaciones y diagramas Los elementos son abstracciones

fundamentales de un modelo; las relaciones ligan estos elementos entre sí; los diagramas agrupan colecciones de elementos.

5.1. ELEMENTOS EN UML

En UML hay cuatro tipos de elementos, los cuales constituyen los bloques básicos de construcción orientados a objetos de este lenguaje de modelado (Booch et al., 1999). Por un lado, los *elementos estructurales* representan las partes estáticas de un modelo, los "materiales de construcción" propios del lenguaje. En total hay siete tipos de elementos estructurales: clases, interfaces, colaboraciones, casos de uso, clases activas, componentes y nodos. Los *elementos de comportamiento* son las partes dinámicas de los modelos UML, son los verbos de un modelo, y representan comportamiento en el tiempo y el espacio. Hay dos tipos principales de elementos de comportamiento: interacciones y máquinas de estados. Los *elementos de agrupación* son las partes organizativas de los modelos UML. El elemento de agrupación principal son los paquetes. Por último, existen *elementos de anotación*, que representan las partes explicativas de los modelos. Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones sobre cualquier elemento del modelo. Hay un tipo principal de elemento de anotación llamado nota.

La intención de este subapartado es la de describir las características de estos elementos, haciendo referencia, siempre que sea posible, a ejemplos mostrados en otros apartados que involucren el uso de los mismos.

Elementos estructurales

Una *clase* es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. La notación gráfica y función de una clase se ha mostrado en anteriores apartados, diferenciándose entre clases de análisis (*véase* apartado 4.2) y clases de diseño (*véase* apartados 2.3.5 y 3.2). Una clase de análisis, aunque tenga una representación gráfica propia asociada a su función (clase de interfaz, clase de entidad y clase de control), puede también adoptar la forma de una clase de diseño.

Al modelar clases, un buen comienzo consiste en especificar las responsabilidades de los elementos del vocabulario. Técnicas como las tarjetas CRC (*véase* apartado 2.3.4.b) y el análisis basado en casos de uso (*véase* apartado 4.1) son especialmente útiles aquí. Al ir refinando los modelos, se proporcionan los atributos y operaciones que mejor satisfagan esas responsabilidades de la clase.

En una clase, es importante especificar la visibilidad de los atributos y operaciones que contenga. Normalmente, los atributos suelen ser privados (símbolo -), sólo accesibles desde los objetos propios de la clase que los contiene, y las operaciones públicas (símbolo +), accesibles por otras clases a través de una referencia a un objeto de la clase que las implementa; sin embargo, tanto unos como otros pueden ser declarados públicos o privados. Además, también se pueden proteger (símbolo #) estas propiedades y funciones de la clase, permitiendo su acceso únicamente desde la propia clase y clases derivadas. Finalmente, se puede indicar que el elemento es visible únicamente para las clases que pertenecen al mismo paquete de agrupación (no se añade símbolo alguno al nombre del atributo u operación).

Además de las opciones de visibilidad, es posible especificar el alcance de un atributo. Puede tener alcance de instancia (normalmente será así), en cuyo caso cada instancia de la clase tiene su propio valor para la característica, o bien, alcance de clase (el atributo aparece subrayado), indicando que sólo hay un valor de la característica para todas las instancias de la clase, es decir, que cualquiera de las instancias de esa clase puede modificar el valor de dicho atributo. El alcance de clase de un atributo se corresponde con lo que en lenguajes como C++ o Java se conoce como atributo estático (*static*). Por su parte, una operación estática es implementada por un método que puede ser llamado aunque no se haya creado ningún objeto de la clase que lo contiene.

Una instancia de una clase es conocida como un *objeto*; dicho de otra manera, si una clase es una abstracción, un objeto es una manifestación concreta de esa abstracción. Esta división común entre clase y objeto se modela de manera distinta. UML distingue un objeto utilizando el mismo símbolo de la clase y subrayando el nombre del objeto.

Casi todos los bloques de construcción de UML presentan este tipo de dicotomía abstracción/instancia. Por ejemplo, se pueden tener casos de uso e instancias de casos de

uso, componentes e instancias de componentes, nodos e instancias de nodos, etc. Concretamente, una instancia es una manifestación concreta de una abstracción a la que se puede aplicar un conjunto de operaciones y que posee un estado que almacena el efecto de estas operaciones. La mayoría de las instancias que se modelan en UML son instancias de clase.

Una *clase activa* es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución, dando origen a actividades de control. Los objetos de una clase activa representan elementos cuyo comportamiento es concurrente con otros elementos. Gráficamente, una clase activa se representa como una clase, pero con líneas más gruesas.

Una *interfaz* es una colección de operaciones que especifican un servicio de una clase o componente. Define un conjunto de especificaciones de operaciones (o sea, sus signaturas), que serán implementadas por las clases o componentes que usen dichos servicios. Gráficamente, una interfaz se representa como un círculo junto con su nombre, aunque también puede adoptar la forma rectangular de una clase (forma expandida); además, raramente se encuentra aislada, sino que suele estar conectada a la clase (*véase* figura 5, apartado 3.2) o componente (*véase* figura 15, apartado 4.4) que la realiza.

Como se dijo en su momento, las interfaces son similares a las clases abstractas (por ejemplo, ninguna de las dos puede tener instancias directas), pero difieren lo bastante como para merecer ser elementos de modelado diferenciados. Una clase abstracta puede tener atributos, pero una interfaz no. Además, las interfaces cruzan los límites del modelo; la misma interfaz, como se ha comentado en el párrafo anterior, puede ser realizada tanto por una clase (una abstracción lógica) como por un componente (una abstracción física que proporciona una manifestación de la clase).

En UML, las interfaces se emplean para modelar las líneas de separación de un sistema. Al declarar una interfaz, se puede enunciar el comportamiento deseado de una abstracción independientemente de una implementación de ella. Además, las interfaces no sólo son importantes para separar la especificación y la implementación de una clase o componente, sino que al pasar a sistemas más grandes, se pueden usar estas interfaces para especificar la vista externa de un paquete o subsistema.

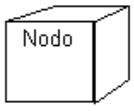
Un *caso de uso* (elipse con borde continuo) es una descripción de un conjunto de secuencias de acciones que un sistema ejecuta y que produce un resultado observable de interés para un actor particular (*véase* apartado 4.1).

Una *colaboración* define una interacción y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. En este sentido, las colaboraciones tienen una dimensión tanto estructural como de comportamiento (pueden ser realizadas, por ejemplo, mediante diagramas de clases y diagramas de interacción). Gráficamente, una colaboración se representa como un elipse con borde discontinuo:

Un caso de uso se especifica normalmente mediante una colaboración (*véase* figura 11, apartado 4.2), que representa la realización de dicho caso de uso. Concretamente, se ha hablado en el capítulo anterior de las realizaciones de caso de uso-análisis y de las realizaciones de caso de uso-diseño.

Un *componente* es una parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la implementación de dicho conjunto. Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos, como clases, interfaces y colaboraciones. Gráficamente, un componente se representa como un rectángulo con pestañas (*véase* figura 15, apartado 4.4).

Un *nodo* es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional (elemento hardware), que por lo general dispone de algo de memoria y, con frecuencia, capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo y puede también migrar de un nodo a otro. Gráficamente, un nodo se representa como un cubo, incluyendo normalmente su nombre (por ejemplo, servidor, cliente, impresora, módem, etc.):



Los nodos son empleados en los diagramas de despliegue, que muestran la configuración de los nodos de procesamiento y los componentes que residen en ellos. En este sentido, los nodos representan el hardware sobre el que se despliegan y ejecutan esos componentes.

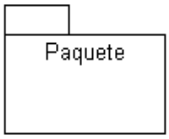
Elementos de comportamiento

Una *interacción* es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular, para alcanzar un propósito específico.

Una *máquina de estados* es un comportamiento que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a estos eventos. El comportamiento de una clase individual o una colaboración de clases puede especificarse con una máquina de estados.

Elementos de agrupación

Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Los elementos estructurales, los elementos de comportamiento, e incluso otros elementos de agrupación pueden incluirse en un paquete. Al contrario que los componentes (que existen en tiempo de ejecución), un paquete es puramente conceptual (sólo existe en tiempo de desarrollo). Gráficamente, un paquete se visualiza como una carpeta:



Los paquetes bien diseñados agrupan elementos cercanos semánticamente y que suelen cambiar juntos. Por tanto, los paquetes bien estructurados son cohesivos y poco acoplados, estando muy controlado el acceso a su contenido. En este sentido, se puede controlar la visibilidad de los elementos contenidos en un paquete del mismo modo que se puede controlar la visibilidad de los atributos y operaciones de una clase (utilizando para ello la misma notación). Normalmente, un elemento contenido en un paquete es público, es decir, es visible a los contenidos de cualquier paquete que importe (*import*, en lenguajes como Java) el paquete contenedor del elemento. Por el contrario, los elementos protegidos sólo

pueden ser vistos por los paquetes derivados (que heredan de otro paquete), y los elementos privados no son visibles fuera del paquete en el que se declaran. El conjunto de las partes públicas de un paquete constituye la interfaz del paquete (elementos exportables).

Cuando un sistema consta de pocas clases que se conocen entre ellas, estamos ante un sistema trivial que no necesita ningún tipo de empaquetamiento. Ahora bien, si son cientos de clases las que se conocen entre ellas, no hay límites para la intrincada red de relaciones que se puede establecer. Además, no hay forma de comprender un grupo de clases tan grande y desorganizado. Este es un problema real para grandes sistemas (Booch et al., 1999), puesto que el acceso simple y no restringido no permite el crecimiento. Para estas situaciones se necesita algún tipo de empaquetamiento controlado para organizar las abstracciones.

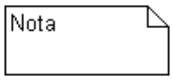
Booch et al. (1999), plantean un ejemplo simple en el que se comprueba las ventajas de organizar los elementos del sistema en paquetes. Dadas dos clases A y B, ambas públicas (se conocen mutuamente) y que están en un mismo paquete, A puede ver a B y B puede ver a A. Sin embargo, si A y B pertenecen a paquetes distintos, aunque sean públicas, ninguna puede acceder a la otra porque sus paquetes contenedores forman un muro opaco. Sin embargo, si el paquete de A importa el paquete de B (relación de dependencia), A puede ver a B (si B es pública), aunque B no puede ver a A (aunque A sea pública). La importación concede un permiso de un solo sentido para que los elementos de un paquete accedan a los elementos de otro. Por tanto, al empaquetar las abstracciones en bloques significativos y luego controlar los accesos mediante la importación, se puede controlar la complejidad de tener un gran número de abstracciones.

La generalización (a la que se hará referencia en el siguiente subapartado) es otro tipo de relación entre paquetes, que permite especificar familias de paquetes (en el mismo sentido que las familias de clases). De hecho, la generalización entre paquetes es muy parecida a la generalización entre clases. En este sentido, al igual que en la herencia de clases, los paquetes pueden reemplazar (redefinir) a los elementos más generales que han sido heredados (elementos públicos y protegidos) y añadir otros nuevos.

Finalmente, dentro de este contexto, hay que hacer mención al concepto de *subsistema*, término que se introdujo en el capítulo anterior para referirse también a la agrupación de elementos en el modelo de diseño y el modelo de implementación. Los subsistemas son similares a los paquetes, pero tienen identidad, es decir, son considerados *clasificadores* (pueden tener instancias, atributos y operaciones, casos de uso, máquinas de estados y colaboraciones, los cuales especifican el comportamiento de ese subsistema). Un subsistema es simplemente una parte del sistema, y se utiliza para descomponer un sistema complejo en partes casi independientes. Un subsistema se representa (en UML) igual que un paquete, añadiendo a su nombre el estereotipo «subsystem». De la misma manera, un sistema también se representa como un paquete estereotipado (con el estereotipo «system»). Sin embargo, a pesar de esta diferencia conceptual establecida por UML entre paquete y subsistema, hay que tener en cuenta, a nivel práctico, que los lenguajes de programación utilizados en la implementación del sistema utilizan un único concepto para referirse a la agrupación de ficheros en compartimentos diferenciados (por ejemplo, los *packages* en Java).

Elementos de anotación

Una *nota* es simplemente un símbolo para mostrar restricciones y comentarios junto a un elemento o una colección de elementos. Gráficamente, una nota se representa como un rectángulo con una esquina doblada, junto con su comentario:



En la figura 5, apartado 3.2, se muestra un ejemplo de uso de este tipo de elemento.

5.2. RELACIONES ENTRE ELEMENTOS

En un apartado anterior se indicaron los cuatro tipos de relaciones disponibles en UML: dependencia, asociación, generalización y realización. En los diagramas UML expuestos a lo largo de este trabajo se han utilizado unas u otras relaciones. La intención de este subapartado es la de detallar cada una de estas relaciones, junto a sus posibles variantes.

Booch et al. (1999), plantean un símil extraído del contexto de la construcción inmobiliaria para concretar en forma de imágenes las relaciones más importantes en el modelado orientado a objetos (dependencias, generalizaciones y asociaciones). Las dependencias son relaciones de uso. Por ejemplo, las tuberías dependen del calentador para calentar el agua que conducen. Las generalizaciones conectan clases generales con otras más especializadas. Por ejemplo, una ventana de patio (clase especializada) es un tipo de ventana (clase general) con hojas de cristal que se abren de lado a lado. Las asociaciones son relaciones estructurales entre instancias. Por ejemplo, las habitaciones constan de paredes y otros elementos; las paredes a su vez pueden contener puertas y ventanas; las tuberías pueden atravesar las paredes. Estos tres tipos de relaciones cubren la mayoría de las formas importantes en que colaboran unos elementos con otros.

Dependencia

Una *dependencia* es una relación semántica entre dos elementos, en la que un cambio en un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (el elemento dependiente). Gráficamente, una dependencia se representa como una línea discontinua dirigida hacia el elemento del cual se depende (figura 17).

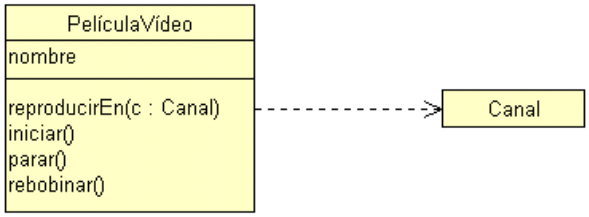


Figura 17. Relación de dependencia entre dos clases

La mayoría de las veces, las dependencias se utilizan en el contexto de las clases, para indicar que una clase utiliza a otra como argumento en la signatura de una operación, tal como se muestra en la figura anterior. Otro ejemplo de uso de relaciones de dependencia se vio en su momento en el contexto de los diagramas de componentes (*véase* figura 15, apartado 4.4). Además, este tipo de relación también se emplea para conectar paquetes.

Si se quieren indicar ciertos matices en las relaciones de dependencia, UML define varios estereotipos (extensión del vocabulario especifico a un problema determinado). En

concreto, en el contexto de los casos de uso, se aplican dos estereotipos a las relaciones de dependencia, el estereotipo *extend* (especifica que el caso de uso destino extiende el comportamiento del caso de uso origen) y el estereotipo *include* (especifica que el caso de uso origen incorpora explícitamente el comportamiento de otro caso de uso). Un ejemplo de uso de estos dos estereotipos lo encontramos en la figura 9, apartado 4.1.

Generalización

Una *generalización* es una relación entre un elemento general (llamado superclase o padre) y un caso más específico de ese elemento (llamado subclase o hijo). La generalización se llama a veces relación «*es-un-tipo-de*»: un elemento (como la clase *Rectángulo*) es un tipo de un elemento más general (por ejemplo, la clase *Figura*). La clase más específica (subclase) comparte la estructura y el comportamiento de la clase más general (superclase), estableciéndose en definitiva una relación de herencia unidireccional (la clase hija hereda de la clase padre). La herencia de atributos y operaciones es un tema que ya se ha tratado en apartados anteriores (apartados 2.3.4.b y 2.3.5). Gráficamente, la generalización o herencia se representa como una línea dirigida contínua con una punta de flecha vacía apuntado al padre (figura 18).

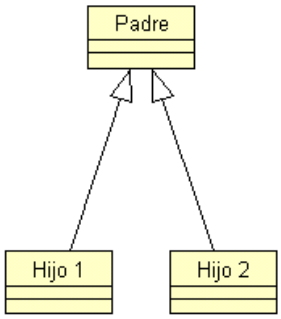


Figura 18. Relación de generalización

Esta relación de generalización se ha mostrado en determinados ejemplos a lo largo de este trabajo (*véase* figura 3, apartado 2.3.5; figura 5, apartado 3.2).

Asociación

Una *asociación* es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. Por defecto, es posible navegar de los objetos de una clase a los de otra, de manera bidireccional. En este sentido, cuando se modela con relaciones de asociación, se están modelando clases que son del mismo nivel; dada una asociación entre dos clases, ambas dependen de la otra de alguna forma, y se puede navegar en ambas direcciones. En cambio, al modelar con relaciones de dependencia o generalización, se están modelando clases que representan diferentes niveles de importancia o diferentes niveles de abstracción, estableciéndose únicamente relaciones unilaterales.

Una asociación, por tanto, especifica un camino estructural a través del cual interactúan los objetos de las clases implicadas. Gráficamente, una asociación se representa como una línea continua que conecta cada uno de sus extremos a una clase distinta (figura 19).

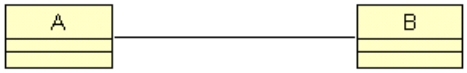


Figura 19. Relación de asociación

Cuando una asociación conecta dos clases, cada una de las clases puede enviar mensajes a la otra en una diagrama de secuencia (*véase* figura 25, apartado 5.3) o en un diagrama de colaboración (*véase* figura 12, apartado 4.2; figura 24, apartado 5.3). Además, también se permite que ambos extremos de la asociación puedan estar conectados a la misma clase (asociación reflexiva), dando a entender que una instancia de la clase está relacionada con otras instancias de la misma clase.

Aparte de la forma básica de la asociación (figura 19), hay ciertos elementos que se le pueden añadir. Por ejemplo, puede tener un nombre, que se utiliza para describir la naturaleza de la relación. Además, cuando una clase participa en una asociación puede jugar un rol específico, es decir, presentar un papel ante la clase del otro extremo; este rol se puede nombrar explícitamente y está relacionado con la semántica de las interfaces. Por ejemplo, según el contexto, una instancia de la clase *Persona* puede jugar el papel de *Madre*, *Cliente*, *Directivo*, *Empleado*, *Cantante*, etc.; cada rol tendrá asociado un conjunto de operaciones distintas (recogidas en una interfaz). En la figura 20, se puede observar que

la clase *Persona* adopta el rol *e*, cuyo tipo es *Empleado* –una interfaz cuya definición incluye una serie de operaciones: *obtenerHistorialEmpleado( )*, *obtenerBeneficios( )*, etc.– Se podría haber añadido a la asociación el nombre o etiqueta *Trabaja para*, aunque el uso de roles, como se puede comprobar, normalmente hace innecesaria la inclusión de este tipo de elemento.

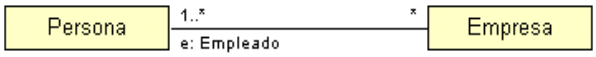


Figura 20. Roles y multiplicidad en una asociación

En esta misma figura aparecen otros descriptores, que hacen referencia a la multiplicidad del rol en la asociación. La multiplicidad indica cuántas instancias de una clase están relacionadas con una única instancia de otra clase; en otras palabras, se especifica que para cada objeto de la clase en el extremo opuesto debe haber tantos objetos en este extremo. En este caso, se observa que una instancia de una empresa puede tener una o más personas empleadas (símbolo 1..*); a su vez, un empleado puede ser contratado por muchas empresas (símbolo *).

Como se ha dicho antes, la navegación a través de una asociación es bidireccional. Sin embargo, puede haber circunstancias en las que se desee limitar la navegación a una sola dirección. Por ejemplo, como se puede ver en la figura 21, al modelar los servicios de un sistema operativo, se encuentra una asociación entre objetos *Usuario* y *Clave*. Dado un usuario, se pueden encontrar las correspondientes claves; pero dada una clave, no se podrá identificar al usuario correspondiente. Se puede representar de forma explícita la dirección de la navegación con una flecha que apunte en la dirección del recorrido.

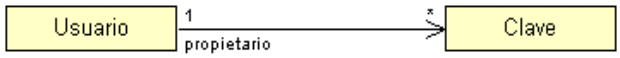


Figura 21. Asociación unidireccional

Una variante de la asociación es la relación de *agregación* entre clases. Si bien una asociación normal entre dos clases representa una relación estructural entre iguales, esta variante permite asociar clases que conceptualmente se encuentran en distinto nivel. Una clase representa el “todo”, que consta de elementos (clases) más pequeños (las “partes”); en términos concretos, un objeto del todo tiene objetos de cada una de las partes asociadas

(figura 22). En otros palabras, se puede decir que la agregación representa una relación del tipo «tiene-un».

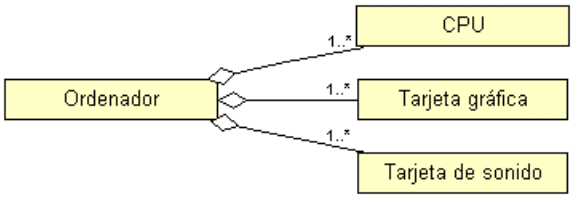


Figura 22. Relación de agregación

La agregación se especifica, como se observa en la figura, añadiendo a una asociación normal un rombo vacío en la parte del todo.

Existe una variante de la agregación (agregación simple), conocida como *composición* (agregación compuesta). Si bien la agregación simple no hace más que distinguir un "todo" de una "parte", la composición establece una fuerte relación de pertenencia, en el sentido que esta relación implica que el "todo" no puede existir sin sus "partes". La composición se especifica añadiendo a una asociación normal un rombo relleno en la parte del elemento compuesto (figura 23).

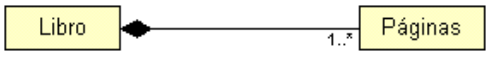
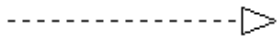


Figura 23. Relación de composición

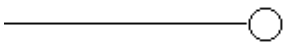
En el ejemplo de agregación (figura 22), si eliminamos la tarjeta de sonido, el ordenador sigue siendo un ordenador. Sin embargo, un libro no es un libro sin sus páginas, esto es, un libro está compuesto de páginas, y precisamente la parte compuesta (extremo con rombo) es responsable de la distribución de las partes (extremo sin rombo), lo que significa que debe gestionar la creación y destrucción de las mismas. En este sentido, los objetos que representan el "todo" y los objetos que representan sus "partes" son creados y destruidos al mismo tiempo. Un ejemplo de composición en el contexto de la simulación estadística se mostró en su momento en el apartado 3.2 (figura 5).

Realización

Una *realización* es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Normalmente, una realización se representa como una línea dirigida discontinua, con punta de flecha vacía que apunta al clasificador que especifica el contrato (notación canónica):



Otra manera de representar una realización es mediante una línea continua no dirigida conectada a un círculo (que representa una interfaz):



Normalmente, este última notación se suele emplear en los diagramas de componentes (*véase* figura 15, apartado 4.4), para unir un determinado componente con la interfaz que le ofrece unos determinados servicios. Por otro lado, la notación canónica es la que se suele emplear para representar la realización de una interfaz por parte de una clase. Además, en este caso, la interfaz suele mostrarse en forma expandida (misma notación gráfica que la clase, y con el estereotipo «interface»), detallando los servicios que ofrece a la clase (*véase* figura 3, apartado 2.3.5; figura 5, apartado 3.2).

Por último, también se utiliza la realización para especificar la relación entre un caso de uso y una colaboración que realiza ese caso de uso, empleando para ello la notación canónica (*véase* figura 11, apartado 4.2). Sin embargo, como la mayoría de veces un caso de uso es realizado por una colaboración, normalmente no se muestra de forma explícita esta relación.

5.3. DIAGRAMAS DE ELEMENTOS

Un *diagrama* es la representación gráfica de un conjunto de elementos y sus relaciones. Los diagramas se utilizan para visualizar un sistema desde diferentes perspectivas. En el contexto del software hay cinco vistas complementarias, que son las más importantes para visualizar, especificar, construir y documentar una arquitectura software (*véase* figura 1,

apartado 2.2): la vista de casos de uso, la vista de diseño, la vista de procesos, la vista de implementación y la vista de despliegue. Cada una de estas vistas involucra modelado estructural (aspectos estáticos del sistema) y modelado de comportamiento (aspectos dinámicos del sistema). Individualmente, cada una de estas vistas permite centrar la atención en una perspectiva del sistema para poder razonar con claridad sobre las decisiones.

Cuando se ve un sistema software desde cualquier perspectiva mediante UML, se usan los diagramas para organizar los elementos de interés. En teoría, un diagrama puede contener cualquier combinación de elementos y sus relaciones. En la práctica, sin embargo, sólo surge un pequeño número de combinaciones, las cuales son consistentes con las cinco vistas mencionadas. En este sentido, UML incluye nueve tipos de diagramas diferenciados (Booch et al., 1999): diagrama de clases, diagrama de objetos, diagrama de casos de uso, diagrama de secuencia, diagrama de colaboración, diagrama de estados, diagrama de actividades, diagrama de componentes y diagrama de despliegue. En este subapartado, nos detenemos en mayor o menor medida en cada uno de estos diagramas.

Diagrama de clases

Un *diagrama de clases* presenta un conjunto de clases e interfaces y las relaciones entre ellas. Son los diagramas más comunes en el modelado de sistemas orientados a objetos y se utilizan para describir la vista de diseño estática de un sistema. Los diagramas de clases que incluyen clases activas se utilizan para cubrir la vista de procesos estática de un sistema.

Los diagramas de clases son la base para un par de diagramas relacionados: los diagramas de componentes y los diagramas de despliegue. Además, los diagramas de clases son importantes no sólo para visualizar, especificar y documentar modelos estructurales, sino también para construir sistemas ejecutables, aplicando ingeniería directa e inversa.

En otros apartados de este trabajo se han mostrado ejemplos concretos de diagramas de clases (*véase* figura 3, apartado 2.3.5; figura 5, apartado 3.2; figura 11, apartado 4.2).

Diagrama de objetos

Un *diagrama de objetos* representa una instantánea de las instancias de los elementos encontrados en un diagrama de clases. Los diagramas de objetos cubren la vista de diseño estática o la vista de procesos estática de un sistema como lo hacen los diagramas de clases, pero desde la perspectiva de casos reales o prototípicos.

Por tanto, un diagrama de objetos congela un instante en el tiempo, expresando la parte estática de una interacción, que consiste en los objetos que colaboran pero sin ninguno de los mensajes enviados entre ellos. Es en la parte dinámica de esta interacción donde se muestran estos mensajes, vista que se recoge en los diagramas de interacción (diagramas de colaboración y diagramas de secuencia).

Los objetos se conectan mediante enlaces, que representan instancias de las relaciones de asociación que se dan entre las clases de dichos objetos. Estos enlaces permiten introducir en los diagramas de interacción una secuencia dinámica de mensajes.

En los diagramas de objetos (y en los diagramas de interacción), el nombre del objeto va subrayado y asociado a la clase a la que pertenece, siguiendo la notación nombre: Clase (también es válido incluir objetos anónimos, es decir, sin nombre).

Por otro lado, los objetos pueden mostrar el valor de sus atributos cuando se considere necesario. Los tipos de estos atributos son definidos en la clase a la que pertenece el objeto.

Diagramas de interacción

Un *diagrama de interacción* muestra una interacción, que consta de objetos y sus relaciones (enlaces), incluyendo los mensajes que pueden ser enviados entre ellos. Los diagramas de interacción cubren la vista dinámica de un sistema. Tanto los diagramas de colaboración como los diagramas de secuencia son un tipo de diagramas de interacción. Un *diagrama de colaboración* es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes. Un *diagrama de secuencia*, en cambio, resalta la ordenación temporal de estos mensajes. Se detallan a continuación.

Diagrama de colaboración

Un *diagrama de colaboración* destaca la organización de los objetos que participan en una interacción (figura 24; figura 12, apartado 4.2). Este diagrama se construye colocando en primer lugar los objetos que participan en la colaboración como nodos de un grafo, a continuación se representan los enlaces que conecta esos objetos como arcos del grafo, y por último, estos enlaces se adornan con los mensajes que envían y reciben los objetos. Precisamente, este último paso se obvia en la representación de los diagramas de objetos (al tratarse de diagramas estáticos).

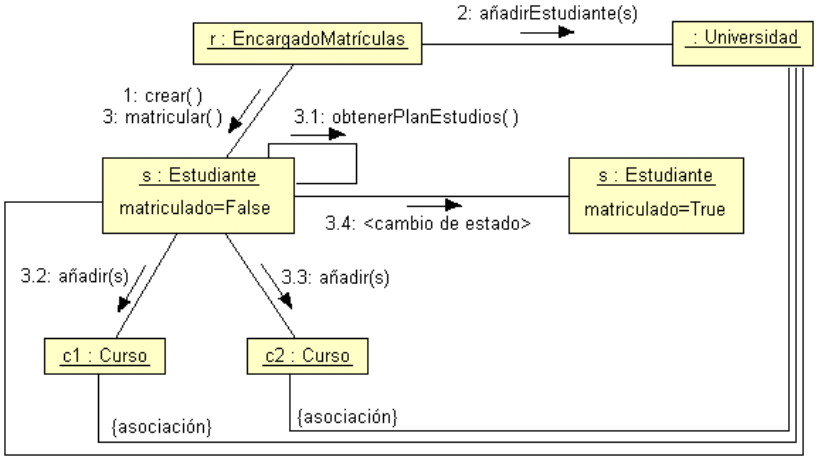


Figura 24. Diagrama de colaboración (Fuente: Booch et al., 1999)

Como se observa en la figura 24, para indicar la ordenación temporal de un mensaje, éste viene precedido de un número (comenzando con el mensaje número 1), que se incrementa secuencialmente por cada nuevo mensaje en el flujo de control (2, 3, etc.). Para representar el anidamiento, se utiliza la numeración decimal de Dewey (1 es el primer mensaje; 1.1 es el primer mensaje dentro del mensaje 1; 1.2 es el segundo mensaje dentro del mensaje 1, etc.). Además, a través de un mismo enlace se pueden mostrar varios mensajes (en la misma o distinta dirección), y cada uno tendrá un número de secuencia único. Otra opción contemplada en un diagrama de interacción es que un objeto dado se envíe un mensaje a sí mismo (mensaje 3.1 en la figura), a través de un enlace que empiece y acabe en él.

En este ejemplo de diagrama de colaboración (figura 24), se especifica un flujo de control para matricular un nuevo estudiante en una universidad, destacando las relaciones

estructurales entre los objetos. La acción comienza cuando el objeto *r* (de la clase *EncargadoMatrículas*) crea un objeto *s* (de la clase *Estudiante*), añade el estudiante a la universidad (mensaje *añadirEstudiante*), y a continuación dice al objeto *Estudiante* que se matricule (mensaje número 3). El objeto *Estudiante* (denominado *s*) invoca la operación *obtenerPlanEstudios* (mensaje 3.1) sobre sí mismo, de donde presumiblemente obtiene los objetos *Curso* (*c1* y *c2*) en los que se debe matricular. Después, el objeto *Estudiante* se añade a sí mismo a cada objeto *Curso*. El flujo acaba con *s* representado de nuevo, mostrando que ha actualizado el valor de su atributo *matriculado*.

La ingeniería directa (creación de código a partir de un modelo) es posible tanto para los diagramas de secuencia como los de colaboración, especialmente si el contexto del diagrama es una operación. Por ejemplo, con el diagrama de colaboración anterior, una herramienta de ingeniería directa podría generar el siguiente código Java para la operación *matricular( )*, asociada a la clase *Estudiante*:

```
public void matricular( )  {
      ColeccionDeCursos c = obtenerPlanEstudios( );
      for (int i = 0; i < c.size( ); i++)
            c.item(i).añadir(this);
      this.matriculado = true;
}
```

Diagrama de secuencia

Un *diagrama de secuencia*, como se ha dicho, es un diagrama de interacción que destaca la ordenación temporal de los mensajes. Gráficamente, un diagrama de secuencia (figura 25) es una tabla que representa objetos, dispuestos a lo largo del eje X, y mensajes, ordenados según se suceden en el tiempo, a lo largo del eje Y.

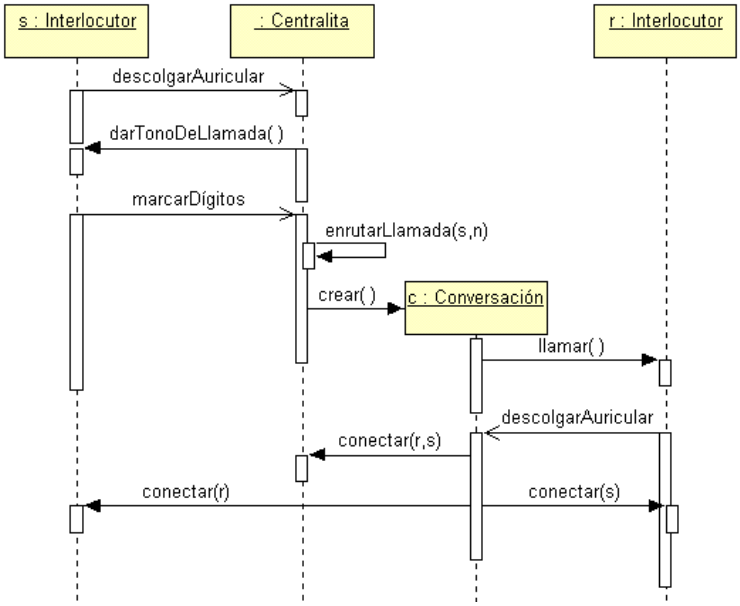


Figura 25. Diagrama de secuencia (Fuente: Booch. et al., 1999)

En este ejemplo, se representa un diagrama de secuencia que especifica el flujo de control para iniciar una simple llamada telefónica entre dos partes. La secuencia comienza cuando un *Interlocutor* (objeto *s*) emite una señal (*descolgarAuricular*) al objeto *Centralita* (objeto anónimo). A su vez, la *Centralita* llama a la operación *darTonoDeLLamada* sobre este *Interlocutor*, y el *Interlocutor* itera sobre el mensaje *marcarDígitos*. El objeto *Centralita* se llama a sí mismo con el mensaje *enrutarLLamada*. A continuación, crea un objeto *Conversación* (objeto *c*), al cual delega el resto del trabajo. El objeto *c* llama a *r* (otro *Interlocutor*), el cual envía de manera asíncrona el mensaje *descolgarAuricular*. Entonces, el objeto *Conversación* indica a la *Centralita* que debe *conectar* la llamada, y luego indica a los dos objetos *Interlocutor* que pueden *conectar*, tras lo cual pueden intercambiar información.

Siguiendo con el ejemplo, en este diagrama se pueden distinguir dos tipos de mensajes, en función del tipo de evento asociado al mensaje. Un evento es la especificación de un acontecimiento que ocupa un lugar en el tiempo y en el espacio; en este sentido, la recepción de un mensaje por parte de un objeto puede considerarse un evento. Se observan en este contexto dos tipos de eventos: eventos de señal y eventos de llamada.

Un evento de *señal* representa la ocurrencia de una señal que es enviada de manera asíncrona por un objeto y es recibido por otro (como es el caso de la señal

descolgarAuricular, que envía el objeto *Interlocutor* al objeto *Centralita*). Si el mensaje es una señal, por tanto, el emisor y el receptor no se sincronizan: el emisor envía la señal pero no espera una respuesta del receptor.

Por otra parte, un evento de *llamada* representa la invocación de una operación y es, en general, síncrono. Ante un evento de llamada síncrono, el emisor y el receptor están sincronizados durante la duración de la operación, de manera que el flujo de control del emisor se bloquea con el flujo de control del receptor, hasta que se lleva a cabo la actividad de la operación. En este sentido, cuando un objeto invoca una operación sobre otro objeto que tiene una máquina de estados[1], el control pasa del emisor al receptor, el evento dispara la transición, la operación acaba, el receptor pasa a un nuevo estado y el control regresa al emisor.

Cuando se pasa un mensaje a un objeto, la acción resultante es una instrucción ejecutable que constituye una abstracción de un procedimiento computacional y que puede provocar un cambio de estado en ese objeto. El tipo más común de mensaje que se modela en un diagrama de interacción es la llamada (representada con una línea continua y punta de flecha rellena), que como se ha dicho invoca una operación de otro objeto (o de él mismo).

Diagrama de casos de uso

Un *diagrama de casos de uso* permite modelar la interacción entre el sistema y los usuarios del mismo. Este diagrama muestra un conjunto de casos de uso, actores y sus relaciones, y se utiliza para describir la vista de casos de uso estática de un sistema. Los diagramas de casos de uso son especialmente importantes para organizar y modelar el comportamiento de un sistema, y como se vio en su momento, se utilizan durante la etapa de captura de requisitos del sistema (figura 9, apartado 4.1).

Un diagrama de casos de uso permite ver el sistema entero como una caja negra; se puede ver qué hay fuera del sistema y cómo reacciona a los elementos externos, pero no se puede

[1] Una máquina de estados puede ser modelada mediante diagramas de estados y diagramas de actividades, detallados más adelante.

ver cómo funciona por dentro; son otras vistas del sistema las que proporcionan esa información de funcionamiento interno.

Diagrama de estados

Un *diagrama de estados* muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Los diagramas de estados cubren la vista dinámica de un sistema, destacando el flujo de control entre estados, es decir, los estados potenciales de los objetos y las transiciones entre esos estados. Gráficamente, un estado se representa como un rectángulo con las esquinas redondeadas y una transición como una línea continua dirigida (*véase* figura 10, apartado 4.1; figura 14, apartado 4.3). Estos diagramas son especialmente importantes en el modelado del comportamiento de una interfaz, una clase o una colaboración (realización de un caso de uso), y resaltan el comportamiento dirigido por eventos de un objeto.

Por tanto, mientras que una interacción modela una sociedad de objetos que colaboran para llevar a cabo alguna acción, una máquina de estados modela la vida de un único objeto, bien sea una instancia de una clase, un caso de uso o un sistema completo.

Una máquina de estados, como se definió en su momento, especifica las secuencias de estados por las que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con sus respuestas a esos eventos. En este sentido, un objeto puede estar expuesto a varios tipos de eventos, tales como una señal, la invocación de una operación (llamada), la creación o destrucción del objeto, el paso del tiempo o el cambio en alguna condición. Como respuesta a estos eventos, el objeto reacciona con alguna acción, que produce un cambio en el estado del objeto o devuelve algún valor.

Un *estado* es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. Un *evento* es la especificación de un acontecimiento significativo que ocupa un lugar en el tiempo y en el espacio. En el contexto de las máquinas de estados, un evento es la aparición de un estímulo que puede activar una transición de estado. Una *transición* es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas

condiciones determinadas. Una *actividad* es una ejecución en curso, dentro de una máquina de estados.

Diagrama de actividades

Un *diagrama de actividades* es un tipo especial de diagrama de estados que muestra el flujo de actividades dentro de un sistema (figura 26).

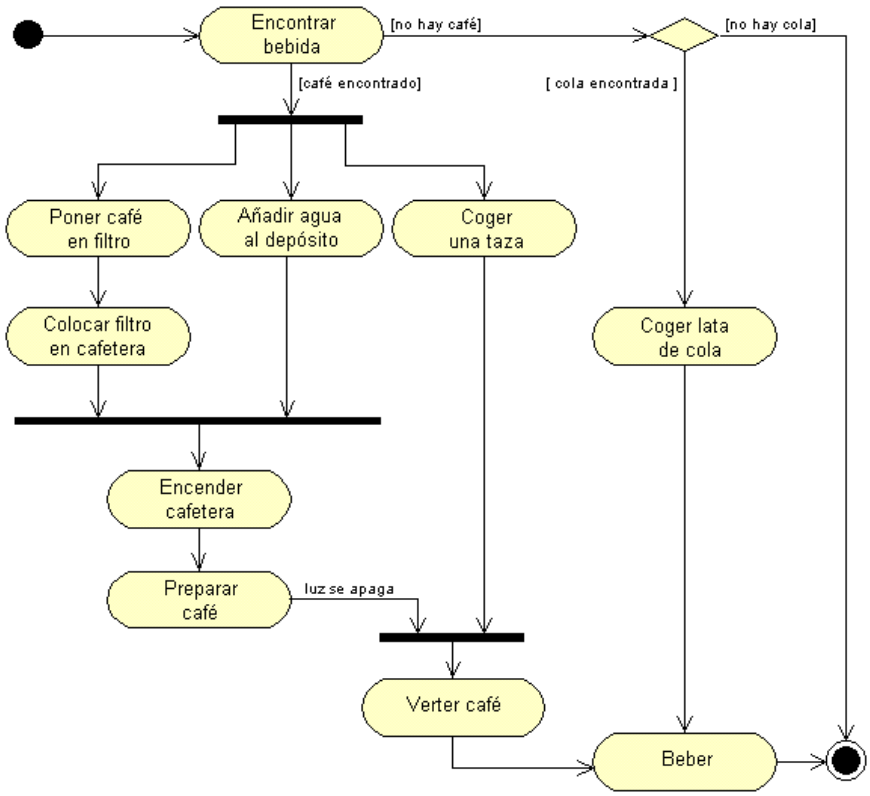


Figura 26. Diagrama de actividades (Fuente: OMG, 2003)

Concretando aún más, se puede decir que un diagrama de actividades es un caso especial de máquina de estados en la que los diferentes estados reflejan estados de actividad, actividades que tienen lugar dentro del objeto (una instancia de una clase, un caso de uso, o un sistema completo). Las actividades producen alguna acción, compuesta de computaciones ejecutables que producen un cambio en el estado del sistema o el retorno de un valor.

Diagrama de componentes

Un *diagrama de componentes* muestra la organización y las dependencias entre un conjunto de componentes. Es un diagrama que permite representar la estructura de ciertos aspectos físicos de los sistemas orientados a objetos, concretamente, la organización y dependencias entre los elementos físicos que residen en un nodo, es decir, ficheros que representan el empaquetamiento físico de clases, interfaces y colaboraciones.

Los diagramas de componentes se utilizan para modelar la vista de implementación estática de un sistema, teniendo en cuenta que un diagrama de componentes individual es normalmente sólo una presentación particular de dicha vista. Esto significa que un único diagrama de componentes no necesita capturarlo todo sobre la vista de implementación del sistema. Un ejemplo de diagrama de componentes se muestra en el apartado 4.4 (figura 15).

Diagrama de despliegue

Un *diagrama de despliegue* muestra la configuración de nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Se relacionan con los diagramas de componentes en que un nodo incluye, por lo común, uno o más componentes. Los nodos representan el hardware sobre el que se despliegan y ejecutan esos componentes.

Los diagramas de despliegue se utilizan para modelar la vista de despliegue estática de un sistema, que implica modelar la topología de hardware sobre el que se ejecuta el sistema. En este sentido, si se desarrolla un software que reside en una máquina e interactúa sólo con dispositivos estándar en esa máquina, que ya son gestionados por el sistema operativo (por ejemplo, el teclado, la pantalla y el módem de un ordenador personal), se pueden ignorar los diagramas de despliegue. Por otro lado, si se desarrolla un software que interactúa con dispositivos que normalmente no gestiona el sistema operativo o si el sistema está distribuido físicamente por varios procesadores, entonces la utilización de los diagramas de despliegue ayuda a razonar sobre la correspondencia entre el software y el hardware del sistema.

Los distintos nodos implicados en un diagrama de despliegue se conectan a partir de relaciones de asociación (figura 27). En este contexto, las conexiones representan enlaces

físicos (normalmente bidireccionales), como es el caso de una conexión directa mediante cable o indirecta por vía satélite.

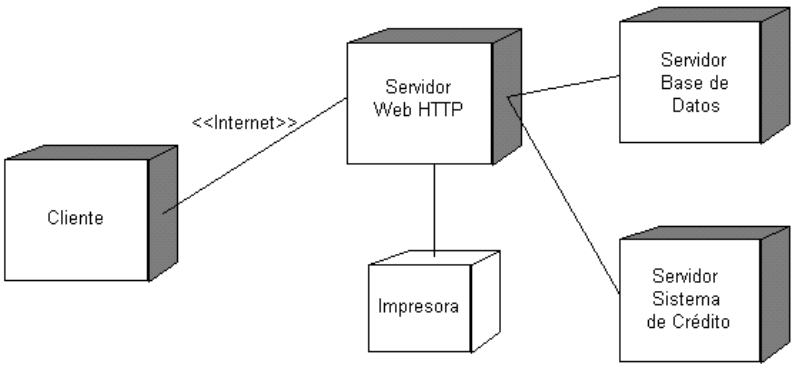


Figura 27. Diagrama de despliegue

Un diagrama de despliegue también puede contener componentes, cada uno de los cuales debe residir en algún nodo. En ese caso, también se incluyen en el diagrama las relaciones de dependencia que existan entre dichos componentes.

PARTE II. PARTE EMPÍRICA

6. *FRAMEWORK* PARA EL MODELADO EN MLG

En este segundo bloque, se desarrolla el análisis y diseño orientado a objetos de un *framework* –plantilla de aplicación reutilizable- para el modelado estadístico con MLG.

Se analiza el contexto del MLG en este primer apartado, con la finalidad de explorar la terminología, métodos, propiedades, algoritmos, funciones y cualquier otro concepto útil o necesario para diseñar posteriormente la arquitectura del *framework*; se indican también los objetivos o requisitos de este *framework* circunscrito al dominio del MLG.

En el siguiente apartado de este bloque (capítulo 7) se plantea el análisis orientado a objetos (AOO) del *framework*, con la captura de requisitos del sistema en forma de casos de uso. Los casos de uso de este sistema y sus relaciones conforman el modelo de casos de uso. Planteamos una propuesta de diagrama de casos de uso arquitectónicamente significativos para el sistema, y, por tanto, fundamentales para el diseño posterior del *framework*.

El diseño orientado a objetos (DOO) del *framework* se expone en el último apartado (capítulo 8). Se consideran los distintos casos de uso de la arquitectura para extraer las clases significativas del sistema; estas clases y sus relaciones formarán parte de los distintos patrones de diseño que conformarán el *framework*.

6.1. EL CONTEXTO DEL MLG

El *Modelo Lineal Generalizado* (MLG) es, sin duda, una de las contribuciones más importantes realizadas en el campo de la estadística en las últimas décadas del siglo XX (Grupo ModEst, 2000b). El MLG, introducido inicialmente por J. Nelder y R.W. Wedderburn en 1972, y desarrollado posteriormente por McCullagh y Nelder (1983, 1989), constituye la generalización natural de los Modelos Lineales clásicos, e incluye como casos particulares la regresión lineal, el análisis de la variancia, el análisis de la covariancia, la regresión de Poisson, la regresión logística, la regresión logit, los modelos log-lineales, los modelos de respuesta multinomial, así como ciertos modelos de análisis de la supervivencia y de series temporales.

El propósito del análisis del contexto MLG en este apartado, no es el de ofrecer una guía sobre los aspectos teóricos y prácticos a considerar en el proceso del modelado estadístico desde una perspectiva explicativa; para ello, existen en la literatura trabajos actuales que recogen esta visión y que proporcionan de manera exhaustiva el acceso a las fuentes originales del MLG (*véase* Grupo ModEst, 2000a, 2000b). La intención, en cambio, es principalmente descriptiva, se centra en explorar el campo o dominio del MLG, para detectar los conceptos clave de la arquitectura del sistema que se va a diseñar. Estos conceptos van a ser necesarios para establecer el marco de una aplicación adecuada para el análisis estadístico con MLG, cuyo diseño permita además una fácil ampliación de su funcionalidad; es lo que se conoce como un *framework*.

El *modelado estadístico* se considera una herramienta fundamental en el estudio de la variabilidad de un conjunto de datos observados, que permite derivar modelos que representen de la manera más óptima posible la relación entre una variable de respuesta y un número de variables explicativas, minimizando su error (discrepancia respecto a los datos observados). En este sentido, se ha de distinguir entre el uso de modelos con fines explicativos, esto es, determinar si realmente existe tal relación (gracias al conocimiento que se tiene sobre el efecto de las variables introducidas en el modelo y de sus interacciones), y el uso de modelos con fines predictivos, que se centra en la selección de las variables que expliquen el mayor porcentaje de variabilidad de la respuesta, con tal de mejorar la predicción del valor de respuesta a partir de las variables explicativas; esta última perspectiva prima los criterios estadísticos (implementados en forma de algoritmos automáticos de selección) sobre los criterios teóricos (en los que el proceso de selección debe ser guiado por el investigador).

Las etapas del modelado estadístico en el contexto del MLG (Grupo ModEst, 2000a) se enmarcan dentro de un proceso de carácter iterativo. Se resumen a continuación:

1. **Especificación:** selección de los modelos más relevantes para describir las características principales de las variables de respuesta. Esto implica tomar decisiones que conciernen a la formulación del componente sistemático del modelo, a los supuestos sobre el componente aleatorio y a cómo los dos componentes son combinados en el modelo (función de enlace).

2. **Selección:** tras la especificación de un modelo particular se requiere estimar los parámetros del componente sistemático y valorar la discrepancia entre los datos observados y los ajustados por el modelo, esto es, valorar el ajuste adecuado del modelo a los datos a partir de criterios de bondad de ajuste y parsimonia. Por tanto, esta etapa representa la fase de estimación y ajuste del modelo, y se enmarca dentro del nivel estadístico-analítico del proceso de investigación científica propuesto por Arnau (1989). En definitiva, se trata de seleccionar el modelo más adecuado para representar a los datos.
3. **Evaluación:** en esta etapa se evalúa si el modelo ajustado es un modelo válido, más allá de que presente un ajuste adecuado a los datos. Se refiere a la adecuación de los aspectos implicados en la etapa de especificación, esto es, evaluar posibles errores de especificación del componente sistemático, de la distribución de probabilidad del componente aleatorio y de la relación asumida entre ambos componentes. Por otra parte, se evalúa también la presencia de observaciones extremas o influyentes, que pueden perturbar las estimaciones obtenidas en la fase de estimación y ajuste.
4. **Interpretación:** en esta etapa se cierra el proceso de modelado, una vez seleccionado el modelo más óptimo en función de los criterios de bondad y ajuste y parsimonia, y tras evaluarlo y decidir que es válido, se ha de proceder a su interpretación e integración en el marco teórico desde el que fue propuesto, esto es, se requiere retornar al nivel teórico-conceptual del proceso metodológico (Arnau, 1989). En los modelos predictivos, esta fase concierne al uso del modelo para realizar predicciones sobre nuevas observaciones que no han sido consideradas en el proceso de selección del modelo.

En la *especificación* de un modelo estadístico se indica qué características de los datos son importantes para el modelado y qué otras no necesitan ser tenidas en cuenta; esta actividad se centra en determinar qué variables explicativas incluir y cuales ignorar, postulando las relaciones matemáticas y probabilísticas entre las variables explicativas y la variable de respuesta, y estableciendo los criterios adecuados de ajuste (Gill, 2000). En este sentido, afirma el autor, la especificación del modelo por parte del investigador es más arte que ciencia, si se tiene en cuenta el enorme número de posibles especificaciones asociadas

incluso a pequeños conjuntos de factores. Generalmente, sin embargo, el investigador suele disponer de justificaciones teóricas sobre algún conjunto de especificaciones; además, en muchos campos existen convenciones sobre la inclusión de variables.

El MLG constituye una extensión del modelo lineal general clásico, que incluye, además de los modelos con componente aleatorio distribuido normalmente, aquellos cuyo componente aleatorio pertenece a la familia exponencial de distribuciones (por ejemplo, además de la distribución normal, las distribuciones binomial y de Poisson). En este sentido, un elemento clave dentro del modelo lineal generalizado es la *función de enlace*[1], función que permite relacionar el componente sistemático (predictor lineal) del modelo con el componente aleatorio del mismo, sea o no lineal. De hecho, el empleo de esta función de enlace permite que el modelo se ajuste a los datos, independientemente de la naturaleza de los mismos, evitando así que sean los datos los que se deban ajustar al modelo (bajo el contexto del Modelo Lineal General).

La distinción entre *valor esperado* y *valor predicho* permite entender el papel de esta función de enlace. Un valor predicho (η_i) por un modelo representa aquel valor obtenido a partir de la aplicación del predictor lineal (componente sistemático) a los datos recogidos en las variables explicativas: $\eta_i=\beta_0+\beta_1 X_i$ (expresa la combinación lineal de las variables explicativas). En cambio, un valor esperado (μ_i) viene dado por la esperanza matemática de las realizaciones de la variable de respuesta aleatoria, E(y). Si bien en el modelo de regresión lineal resulta que $\eta_i=\mu_i$, es decir, el resultado del predictor lineal proporciona directamente los valores esperados, en modelos no lineales la igualdad anterior no se cumple, puesto que el valor predicho se encuentra medido en una escala lineal con rango entre $-\infty$ y $+\infty$, mientras que el valor esperado de una variable cuyo componente no es lineal, se encuentra medido en la escala propia de la variable de respuesta.

[1] Como señala Krzanowski (1998, p. 168), para un problema particular se pueden plantear distintas funciones de enlace, por lo que el investigador debe decidir cuál de ellas es la más apropiada en cada caso; sin embargo, se puede introducir una simplificación en este proceso si se utiliza como función de enlace aquella que define al parámetro canónico de la distribución y que, por este motivo, se denomina «función de enlace canónica» (Grupo ModEst, 2000b, p. 8).

Por ejemplo, en el modelado de una variable de respuesta de tipo recuento, el valor esperado μ_i sólo puede tomar valores enteros iguales o superiores a 0, por tanto, el valor esperado y el valor predicho se hallan en diferentes escalas. La función de enlace interviene en este punto, transformando el valor de recuento esperado a la escala del predictor lineal $(-\infty, +\infty)$: $g(\mu_i)=\eta_i$. Por su parte, la inversa de la función de enlace realiza el proceso inverso, ya que al aplicarla al resultado del predictor lineal se obtienen el valor esperado, que se halla en la escala de la variable de respuesta: $g^{-1}(\eta_i)= \mu_i$.

En el modelo de regresión lineal, puesto que $\eta_i=\mu_i$ (los valores predichos y los valores esperados se hallan en la misma escala), se deduce que la función de enlace en este tipo de modelos es la función identidad: $g(\mu_i)= \mu_i$. En el modelado de datos de recuento, que siguen una distribución de Poisson, la función de enlace que transforma el valor esperado a la escala del predictor lineal es el logaritmo: $\log(\mu_i)=\eta_i$, que representa la función que define al parámetro canónico de la distribución (Grupo ModEst, 2000b, p. 263), mientras que la función exponencial (inversa de la función de enlace logarítmica), permite obtener el valor esperado por el modelo a partir del valor predicho: $\exp(\eta_i)= \mu_i$.

Dentro del proceso de modelado estadístico de datos, la fase de *selección* sigue a la de especificación, e integra la estimación de parámetros y el ajuste de modelos. En el contexto del MLG, la estimación máximo-verosímil de los parámetros β_i del predictor lineal del modelo se realiza mediante el algoritmo denominado «Mínimos cuadrados iterativamente ponderados» (*Iterative Weighted Least Squares*, IWLS). Nelder y Wedderburn (1972), mostraron que las estimaciones de máxima verosimilitud para todos los modelos incluidos en los *Modelos Lineales Generalizados* podrían ser obtenidos a través de este algoritmo, que proviene de utilizar el método de Fisher-Scoring, en el que en cada iteración se lleva a cabo una regresión mediante mínimos cuadrados ponderados, y que además, es equivalente al algoritmo de Newton-Raphson cuando se emplea la función de enlace canónica. Estos algoritmos se enmarcan dentro del método de estimación conocido como de «Máxima Verosimilitud» (*Maximum Likelihood*, ML), adecuado para los modelos con distribución del componente aleatorio conocida. Por otro lado, el método de estimación generalmente utilizado para el modelo de regresión lineal (componente aleatoria normal) es el método de «Mínimos Cuadrados Ordinarios» (*Ordinary Least Squares*, OLS).

En los modelos de regresión con componente aleatoria normal y con variancia constante, los estimadores máximo verosímiles y los estimadores mínimo cuadráticos coinciden (Long, 1997). En este sentido, el método OLS es un caso particular del procedimiento de estimación IWLS.

El algoritmo IWLS consiste en los siguientes pasos (Grupo ModEst, 2000b, p. 12):

1. Obtener una primera estimación de los parámetros. Para ello se aplica la función de enlace a los valores observados, y el vector resultante *G* se regresa sobre las variables explicativas X, empleando el método de mínimos cuadrados ordinarios:

$$\hat{\beta}_j = (X'X)^{-1}(X'G)$$

2. A partir de las estimaciones obtenidas se calculan los valores predichos $\hat{\eta}_i$ (estimación puntual en una muestra de η_i), que se transforman mediante la inversa de la función de enlace para obtener los valores esperados m_i (estimación puntual en una muestra de μ_i).

3. A partir de los valores predichos y esperados obtenidos en el paso 2, se construye una «variable de respuesta de trabajo» Z_i que refleja la diferencia entre valores esperados y valores observados, y una «matriz de ponderación» W cuya diagonal principal recoge las variancias de los valores esperados m_i.

4. La variable Z_i se regresa sobre las variables explicativas X considerando los valores de ponderación W, de manera que se obtiene una estimación por mínimos cuadrados ponderados mediante:

$$\hat{\beta}_j = (X'WX)^{-1}(X'WZ)$$

5. Se repiten los pasos 2 a 4 iterativamente hasta que se cumpla el criterio de convergencia determinado a priori.

En cuanto al estudio de la significación estadística de los coeficientes del modelo, el contraste de la hipótesis de que un coeficiente $\beta_j=0$ se puede realizar mediante el test de Wald, que viene dado por:

$$z = \frac{bj}{EE(bj)}$$

Una exposición detallada del test de Wald se puede encontrar en Rodríguez (2002). Como exponen Hutcheson y Sofroniou (1999, p. 129), el test de Wald debe utilizarse con precaución, puesto que tiende a exagerar la significación estadística de variables con valores de coeficientes altos, y se ha comprobado además que no es fiable para muestras pequeñas. Debido a estas restricciones, una aproximación más general del estudio de la significación de los coeficientes de un modelo es el uso del test de la razón de verosimilitud, que se basa en la óptica de la comparación de modelos (Grupo ModEst, 2000a, 2000b).

El cálculo del EE de la distribución muestral de un coeficiente b_j es otro de los elementos necesarios para realizar inferencias poblacionales. En este sentido, dicho EE permite calcular el intervalo de confianza (IC) del parámetro β_j para evaluar el efecto real de una variable en la población[2]. Por otra parte, de cara a la interpretación del modelo, el IC alrededor de cualquier valor m_i esperado por el modelo se puede realizar, en la práctica, siguiendo el mismo procedimiento descrito para la estimación por intervalo de los parámetros β_j en presencia de interacción (Grupo ModEst, 2000b, p.19). Además del intervalo de confianza, también es posible calcular un intervalo de predicción (IP) alrededor de m_i que permita conocer el rango de valores entre los que se encontrará y_i para los sujetos de la población origen con una determinada combinación de valores en las variables explicativas X_j (Grupo ModEst, 2000b, p.20).

Una vez estimados los parámetros de un modelo concreto, se ha de valorar su grado de ajuste –*bondad de ajuste* del modelo–, es decir, la magnitud de la discrepancia entre los datos observados y los esperados por el modelo. En este sentido, el cálculo de la discrepancia D de un modelo (conocida generalmente como $-2\log L$) viene determinado

[2] Si el coeficiente está asociado a un término de interacción, Judd y McClelland (1989, p. 260) presentan un procedimiento para obtener el IC de una variable X para cualquier valor de la variable modificadora Z distinto de 0. En general, si se desea obtener el IC del efecto de X para el valor *c* en Z, plantean transformar la variable modificadora Z restando a sus valores originales el valor *c*, y a continuación, regresar Y sobre X y la variable Z «desviada». El EE del coeficiente de regresión de la variable X en este nuevo modelo representa el EE del coeficiente de regresión del término de interacción para un valor concreto *c* de la variable Z.

por la distribución del componente aleatorio de dicho modelo. En un modelo de regresión de Poisson, por ejemplo, la discrepancia viene dada por:

$$D = -2 \times \sum_{i=1}^{n} \left(y_i \times \log\left(\frac{m_i}{y_i} \right) + \left(y_i - m_i \right) \right)$$

La expresión de cálculo de la discrepancia para otras distribuciones pertenecientes a la familia exponencial se puede consultar en Grupo ModEst (2000b, p. 15). La formulación y expresión general del cálculo de la discrepancia de un modelo bajo la familia exponencial de distribuciones se detalla en Rodríguez (2002, apéndice B).

Otras medidas que extienden el análisis de la bondad de ajuste de un modelo, y que permiten la comparación entre modelos, son el índice R^2, los índices *pseudo*-R^2, el *AIC*, el *BIC*, etc. En los modelos de regresión normal, las tres medidas más utilizadas para evaluar la bondad de ajuste son el coeficiente de determinación (R^2), el coeficiente de determinación ajustado ($\bar{R}^2$) y el estadístico C_p de Mallows (1973) (Grupo ModEst, 2000a, p. 215). La familia de índices *pseudo*-R^2 incluye aquellos índices que representan una aproximación al índice R^2 en modelos de regresión no lineal; una propuesta de índice *pseudo*-R^2, para el modelo de regresión de Poisson, la presentan Cameron y Trivedi (1998, p.153). Los índices *AIC* (*Akaike information criterion*) y *BIC* (*Bayesian information criterion*) aparecen descritos, entre otros, en Hardin y Hilbe (2001, p. 45).

En el proceso de ajuste de un modelo, serán considerados un conjunto de modelos que representan aproximaciones alternativas a los datos observados. Lindsey (1997) introduce una terminología para nombrar a cada uno de estos modelos:

- *Modelo saturado* (MS). Se caracteriza porque el número de parámetros que estima el modelo es igual al número de observaciones de que se dispone. El MS reproduce exactamente los datos observados (discrepancia nula), pero tiene poca probabilidad de ser adecuado en replicaciones posteriores del estudio.
- *Modelo máximo*. Es el modelo más complejo que puede ser considerado a partir del número de variables explicativas registradas, incluyendo variables de control que permitan ajustar o mejorar la precisión de las estimaciones y aquellos términos de interacción que se consideren relevantes. Como norma general, se aconseja

establecer un modelo máximo lo más reducido posible, es decir, con términos cuya inclusión esté justificada por los actuales conocimientos teóricos.

- *Modelo mínimo*. Este modelo incluye el conjunto de parámetros mínimo que, por motivos de diseño, debe ser estimado.
- *Modelo nulo* (MN). Este modelo incluye un único parámetro y, por tanto, proporciona un mismo valor esperado para todas las observaciones, que es la media de los valores observados.
- *Modelo cero*. Se trata de un modelo que no estima ningún parámetro, puesto que el valor de estos parámetros queda fijado por el investigador. Este modelo particular es necesario para realizar pruebas de conformidad sobre el valor de un parámetro.
- *Modelo de trabajo* (MT). Es el modelo objeto de comparación en cada paso del proceso de ajuste y selección del modelo final, y se encuentra, en cuanto al número de parámetros que estima, entre el modelo máximo y el modelo mínimo.

El objetivo del modelado es encontrar el modelo que reproduzca de la manera más exacta posible los datos observados con un menor número de parámetros. Este objeto representa un balance entre bondad de ajuste y parsimonia, donde los modelos saturado y nulo forman los dos extremos de un continuo. En este sentido, el MN asigna toda la variabilidad observada en los datos al componente aleatorio del modelo y ninguna al componente sistemático, mientras que el modelo saturado asigna toda la variabilidad observada al componente sistemático.

El procedimiento de selección del «mejor» modelo entre un conjunto de modelos posibles para representar los datos observados se basa en la comparación entre pares de modelos, estrategia básica de reducción del error en el proceso de modelado. En la comparación de modelos se evalúa la significación estadística del aumento o disminución en el componente de error al comparar dos modelos jerárquicos. En este sentido, se aplican técnicas clásicas del contraste de hipótesis, de manera que uno de los modelos se corresponde con la formulación de la H_0 y el otro con la formulación de la H_1. En concreto, este procedimiento de selección se basa en dos estrategias complementarias de comparación de modelos: el *ajuste global* y el *ajuste condicional*.

La definición del ajuste global depende de que se estén modelando datos agrupados o datos individuales; en el primer caso el modelo de trabajo (MT) se compara con el modelo saturado (MS), por tanto, supone comparar los valores esperados con los observados. Con datos individuales (como es el caso de las variables continuas) el MS es inviable en la práctica, por lo que el ajuste global de un MT se realiza comparando dicho modelo con el modelo nulo (MN).

El procedimiento de ajuste condicional consiste en comparar dos modelos anidados, esto es, uno es un caso particular del otro (los términos del modelo menor están incluidos en el modelo que tiene un mayor número de parámetros). El objetivo de este procedimiento es seleccionar el modelo más simple («modelo restringido», MR), cuya discrepancia (diferencia entre los valores observados y los esperados por el modelo) no difiera significativamente de la discrepancia del modelo más complejo en el que esté anidado («modelo ampliado», MA).

La *comparación de dos modelos* se realiza confrontando sus discrepancias mediante la prueba ΔD, denominada también «test de la razón de verosimilitud» (*Likelihood Ratio Test*), puesto que depende del ratio de la verosimilitud de un modelo (L_{MR}) respecto a otro (L_{MA}): $\Delta D = -2\log(L_{MR}/L_{MA})$. En otros términos, esta prueba recoge la diferencia entre las discrepancias $-2\log L$ de ambos modelos (por equivalencia a la expresión anterior). La prueba ΔD de ajuste global para el caso de datos individuales viene dada por la expresión $\Delta D = D_{MN} - D_{MT}$, mientras que la prueba de ajuste condicional se obtiene a través de la expresión $\Delta D = D_{MR} - D_{MA}$. La prueba ΔD tiene una distribución asintótica χ^2 y, por tanto, su significación se obtiene situando el valor obtenido en la distribución χ^2, con grados de libertad iguales a la diferencia entre los grados de libertad de los dos modelos comparados. Un caso particular de esta prueba se da cuando el componente aleatorio de los modelos que se comparan sigue la distribución Normal; en este caso, las discrepancias de ambos modelos se contrastan mediante el estadístico *F* (Grupo ModEst, 2000a).

En el contexto del ajuste condicional de modelos, además de la significación estadística del valor de la diferencia de discrepancias, se emplean otros índices adicionales, como es el caso del índice ΔR^2 (incremento de R^2) y el índice *PRE* (reducción proporcional del error), medidas consideradas útiles para el análisis de regresión lineal múltiple (Grupo Modest,

2000a, p. 159). Estas medidas permiten cuantificar la aportación individual de cada una de las variables del modelo a la explicación de la respuesta, aspecto éste que está fuertemente relacionado con la colinealidad. El estudio de la colinealidad se realiza habitualmente mediante el cuadrado del coeficiente de correlación múltiple (R^2_j) de una variable explicativa obtenido al regresar esta variable sobre todas las demás variables explicativas del modelo, o bien, a partir de las medidas *Tolerancia* o *FIV* (*Factor de inflación de la variancia*), que están relacionadas directamente con R^2_j (Grupo ModEst, 2000a, p. 164).

La etapa de *evaluación* es también fundamental en el proceso de modelado estadístico. Existen variadas técnicas e índices para la evaluación de la «corrección» de un modelo (Grupo Modest, 2000a, cap. 6, 2000b, apéndice D; Hardin y Hilbe, 2001, cap. 4), esto es, para evaluar si las asunciones en relación a la especificación del modelo son correctas y para detectar la posible presencia de observaciones extremas o influyentes que afecten a las estimaciones.

En este sentido, el análisis de residuales es una técnica de evaluación fundamental para valorar la adecuación de un modelo. Los residuales miden la discrepancia entre los valores observados y los valores ajustados (esperados) por el modelo, y se utilizan en general para detectar posibles observaciones influyentes (atípicas, anómalas, inconsistentes,...). La obtención de los residuales *ordinarios* es directa, puesto que representa simplemente la diferencia entre el valor observado y el valor esperado para cada observación. La división del residual ordinario por su desviación estándar da lugar a los residuales *estandarizados*. Otros tipos de residuales, como el «residual estudentizado» o el «residual eliminado» son descritos en Grupo Modest (2000a, 2000b). Hardin y Hilbe (2001) detallan también un conjunto de residuales de interés.

El grado en que una observación concreta afecta a los coeficientes estimados es una medida de influencia (Hardin y Hilbe, 2001). Las principales medidas de influencia en MLG se recogen en Grupo ModEst (2000a, 2000b). El *valor de influencia* (*leverage*) de las observaciones sobre el ajuste del modelo se refleja en los elementos h_{ii} de la diagonal principal de la matriz H («matriz sombrero»). Aquella observación que tenga un valor h_{ii} («valor influyente») alto se dice que tiene «influencia» (*leverage*). Observaciones con un valor de influencia alto requieren especial atención ya que el ajuste puede ser

excesivamente dependiente de ellas. La matriz H permite, por tanto, detectar posibles observaciones influyentes. La influencia también se puede medir por medio de la *distancia de Mahalanobis* (distancia de una observación *i* al centroide), de forma que el valor de esta distancia será grande cuando la influencia también lo sea, y a la inversa.

Otras medidas de influencia permiten evaluar la influencia de una observación sobre la estimación de los coeficientes de regresión. Es el caso de la *distancia de Cook*, y los índices *DFFITS* y *DFBETAS*. La estrategia a seguir consiste en obtener la estimación de los parámetros del modelo con y sin esta observación.

Los métodos informales para la evaluación de modelos descansan en la visualización de representaciones gráficas basadas, fundamentalmente, en los residuales de un modelo. En general, al realizar una representación gráfica de los residuales de un modelo se asume que éste es adecuado cuando la distribución de los residuales no obedece a ningún patrón. Por el contrario, la observación de un patrón concreto indicaría que es posible ajustar un modelo más adecuado a los datos. Al recurrir a métodos gráficos para evaluar un modelo, el objetivo no acostumbra a ser verificar el cumplimiento de un supuesto determinado, sino más bien, valorar si la representación gráfica revela su incumplimiento (Grupo ModEst, 2000a, p. 283). Una relación de los principales gráficos utilizados para la evaluación de modelos, así como el diagnóstico específico que permiten realizar, se recoge en Grupo ModEst (2000b, apéndice D).

El problema de la *sobredispersión* es un tema de evaluación prioritario cuando se usa la distribución binomial o la distribución de Poisson en el ajuste de modelos. De hecho, como señalan McCullagh y Nelder (1989), la sobredispersión es la norma en la práctica y la equidispersión es la excepción. La sobredispersión se evalúa habitualmente, o bien, considerando directamente si la relación media-variancia condicionada se ajusta a la distribución asumida para el modelo, o bien, evaluando la relación entre la discrepancia del modelo ajustado y sus grados de libertad (Grupo ModEst, 2000b): $s = D / gl$, de manera que un valor *s* igual a 1 indica una correcta relación media-variancia (equidispersión), valores superiores son indicativos de presencia de sobredispersión, y valores inferiores a 1 reflejan infradispersión.

La prueba de razón de verosimilitud (*LR*), la prueba de Wald y la prueba multiplicador de Lagrange (*LM*) son consideradas pruebas clásicas en la comparación de modelos anidados, que pueden ser utilizadas, en esta situación, para el diagnóstico de sobredispersión. Una descripción detallada de éstas se encuentra en Vives (2002).

La etapa de *interpretación* del modelo incluye la obtención de los valores predichos por el modelo a partir de su predictor lineal, además de la transformación de dichos valores a la escala de la variable de respuesta (valores ajustados o esperados). También puede resultar de interés conocer la «probabilidad predicha» de valores concretos m_i de la variable de respuesta para distintos valores en la variable explicativa, así como el IC de esta probabilidad predicha (Grupo ModEst, 2000b, p. 35). El cálculo del IC de la probabilidad predicha requiere del «EE predicho» (error estándar del error de predicción), que coincide con el EE del valor esperado m_i.

Una última referencia en relación al modelado estadístico con MLG es el uso de algoritmos de selección automática, utilizados habitualmente en el contexto de las investigaciones de tipo exploratorio o predictivo. Destacan tres métodos de selección, que se basan en la adición o sustracción secuencial de términos en el modelo: el método «backward elimination», el método «forward selection» y el método «stepwise regression» (Hutcheson y Sofroniou, 1999, p. 96; Grupo ModEst, 2000a, p. 217). Un cuarto método, denominado «all possible regressions», parte de la estimación de todas las posibles ecuaciones (Grupo ModEst, 2000a, p. 216).

6.2. REQUISITOS DEL FRAMEWORK

En el subapartado anterior se ha descrito la estructura del contexto MLG en lenguaje natural, con la intención de establecer posteriormente la arquitectura de un sistema circunscrito a dicho contexto –aunque se presentarán ejemplos de extensión de este sistema a otros tipos de modelos como los Modelos Aditivos Generalizados (MAG) (Hastie y Tibshirani, 1990)–; esta arquitectura representa la base del *framework* que se pretende diseñar. El análisis de dicho *framework* (a partir de un modelo de casos de uso) y su

posterior diseño (a partir de un modelo de diseño) es el objetivo de los siguientes apartados.

Antes de enunciar los requisitos que debe cumplir el *framework* MLG, conviene ahondar en el concepto de *framework*, para delimitar de forma clara su papel en el desarrollo de aplicaciones. Johnson y Foote (1988) describen un *framework* como un diseño abstracto orientado a objetos para un determinado tipo de aplicación, que se compone de una clase abstracta para cada componente principal del diseño; contendrá normalmente una librería de subclases que pueden ser utilizadas como componentes del diseño. Por otro lado, Booch et al. (1999) conciben un *framework* como un patrón arquitectónico que proporciona una plantilla extensible para aplicaciones dentro de un dominio.

La palabra *framework* recoge en su definición conceptos familiares, como «clase abstracta», «interfaces», «patrón» y «extensible». Además, un *framework* está asociado al diseño de un determinado tipo de aplicación. El concepto «tipo de aplicación» acota el alcance de un *framework*; no es un diseño de propósito general, sino que está ligado a un dominio concreto (en nuestro caso, al ámbito del modelado estadístico en el dominio MLG). En ese sentido, se usa también el término «plantilla extensible» para dar a entender, por un lado, que se compone de una estructura fija de componentes con una determinada funcionalidad, y por otro lado, que es una estructura común para un tipo determinado de aplicaciones que puede ser ampliada con nuevos componentes. El término «patrón arquitectónico» denota precisamente la reusabilidad de la arquitectura del diseño.

Si bien el concepto de *framework* presenta similitudes con el término *patrón de diseño* (definido en el apartado 3.2), ambos se diferencian en tres aspectos fundamentales (Gamma et al., 2003):

1. *Los patrones de diseño son más abstractos que los frameworks*. Los *frameworks* pueden plasmarse en código (como aplicación específica), pero sólo los ejemplos de los patrones pueden ser plasmados en código. Uno de los puntos fuertes de los *frameworks* es que se pueden escribir en lenguajes de programación y de ese modo ser no sólo estudiados, sino ejecutados y reutilizados directamente. Por el contrario, los patrones de diseño tienen que ser implementados cada vez que se emplean. Los

patrones de diseño también reflejan la intención, las ventajas e inconvenientes y las consecuencias de un diseño.

2. *Los patrones de diseño son elementos arquitectónicos más pequeños que los frameworks*. Un *framework* típico contiene varios patrones de diseño, pero lo contrario nunca es cierto.
3. *Los patrones de diseño están menos especializados que los frameworks*. Los *frameworks* siempre tienen un dominio de aplicación concreto. Sin embargo, los patrones de diseño se pueden usar en prácticamente cualquier tipo de aplicación.

En UML, un *framework* se modela como un paquete estereotipado (con etiqueta «*framework*»). Cuando se mira dentro del paquete se pueden ver mecanismos existentes en cualquiera de las diferentes vistas de la arquitectura de un sistema (Booch et al., 1999). Por ejemplo, no sólo se pueden encontrar colaboraciones parametrizadas (que recogen las clases concretas del diseño y sus relaciones), sino que también se pueden encontrar casos de uso (que analizan el funcionamiento del *framework*), así como colaboraciones simples (que proporcionan conjuntos de abstracciones, como clases abstractas e interfaces, sobre las que se puede construir).

Muchas veces, un *framework* incorpora clases de una o más bibliotecas de clases predefinidas llamadas *toolkits*. Un *toolkit* es un conjunto de clases relacionadas y reutilizables diseñadas para proporcionar funcionalidad útil de propósito general (Gamma et al., 2003). Los *toolkits* no imponen un diseño particular en una aplicación; simplemente proporcionan funcionalidad que puede ayudar a que la aplicación haga su trabajo. Permiten, por tanto, evitar recodificar funcionalidad común. Un *toolkit* de clases para la creación de distintos tipos de gráficos proporcionaría funcionalidad gráfica a cualquier *framework* de análisis de datos, sin necesidad de que éste los diseñe de manera específica en su estructura. En este sentido, los diversos métodos gráficos para evaluar un modelo en contexto MLG podrían estar implementados en un *toolkit* de estas características.

En definitiva, se puede decir que un *framework* consiste en un conjunto de clases cooperantes que forman un diseño reutilizable para un determinado tipo de aplicación. Un *framework* proporciona una guía arquitectónica para dividir el diseño en clases abstractas y definir sus responsabilidades y colaboraciones (Gamma et al., 2003).

El objetivo de este trabajo no es proporcionar un paquete de clases de propósito general, sino crear un patrón arquitectónico para el modelado estadístico en el contexto del MLG, esto es, el diseño de un *framework* para el MLG.

Este *framework* deberá cumplir con un conjunto de requisitos que se enuncian a continuación:

- El diseño del *framework* se basa en el paradigma OO, y sigue las directrices del PU. Por otro lado, con el objetivo de facilitar la comunicación científica, los artefactos derivados de este proceso se ajustan el estándar UML.

  Ciertas características de la OO (encapsulación, herencia, composición de objetos y polimorfismo, fundamentalmente), focalizadas en el uso de patrones de diseño, permiten conseguir una estructura asentada y flexible a los cambios.

  En este sentido, un *framework* debe ser un sistema abierto que permita la integración de nuevos elementos sin necesidad de modificar la implementación de los ya existentes. Se trata de plantear la estructura de un sistema flexible, que tenga en cuenta aquellos objetos que podrían variar en el diseño; en definitiva, que sea posible, por ejemplo, sustituir el algoritmo de estimación de un modelo sin necesidad de retocar la implementación del sistema, o el intercambio directo de la función de enlace asociada al objeto IWLS o del objeto que implementa el cálculo de la discrepancia del modelo, o la extensión de dicho modelo para tomar en consideración procedimientos alternativos para modelar la sobredispersión en modelos de regresión Binomial o de Poisson.

- El *framework* debe admitir un doble uso: el modelado estadístico y la simulación estocástica. Esto implica que su diseño ha de cumplir con dos requisitos:

  - Facilidad de uso. Esta característica es especialmente importante para la vertiente de modelado estadístico. Por ejemplo, los constructores de algunas clases admiten como parámetros cadenas de caracteres para que se desencadene el proceso de construcción de un objeto dado, como es el caso de la construcción de un objeto del tipo *Formula*.

- Máxima eficiencia. Supone la posibilidad de que los objetos ya creados puedan ser reutilizados por el sistema sin necesidad de volver a repetir el proceso de su creación. Esta característica es un requisito fundamental en cualquier procedimiento que requiera de computación intensiva, como es el caso de la simulación estocástica.

- El diseño del *framework* deberá ser totalmente independiente de la interfaz de usuario. En este sentido, debería ser compatible con tres interfaces posibles:
  - Sistema de tipo interfaz de comandos mediante un lenguaje interpretado, al estilo de los paquetes estadísticos SAS, R, S-Plus o el entorno más general Matlab, que puede ser combinado con
  - Interfaz Gráfico de Usuario (IGU), al estilo de paquetes estadísticos como SPSS.
  - Entorno de Desarrollo Integrado (EDI) que facilite al investigador la extensión del sistema. En este sentido, uno de los objetivos fundamentales de este EDI es que asista al investigador en esta tarea, por ejemplo, con ayudas sintácticas o sugerencias de diseño, al estilo de los EDI para lenguajes de programación de propósito general, como Java o C++. Un EDI de estas características facilita la transición del usuario al rol de desarrollador.

- Parte del sistema será diseñado de tal forma que la modificación de ciertos elementos provoque la actualización automática de los elementos vinculados. Por ejemplo, si un objeto de tipo *Formula* modifica su expresión, dicho objeto se encargará de notificar el cambio a los objetos que hacen uso de él, como los objetos *Modelo Estadístico*, para que éstos actualicen su ajuste en función de una nueva expresión de la ecuación estructural del modelo.

7. MODELO DE CASOS DE USO

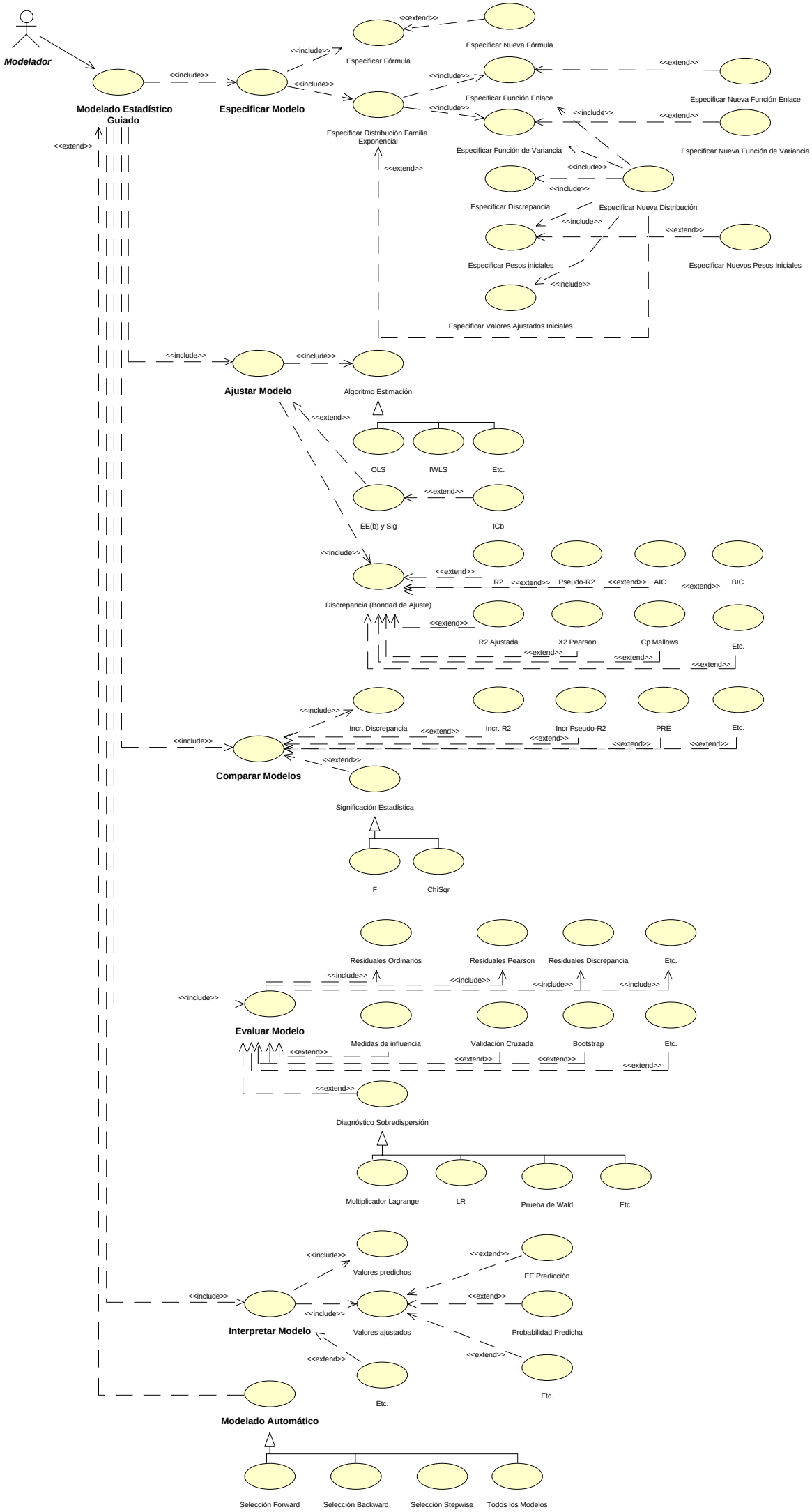
El modelo de casos de uso constituye, especialmente en aquellos casos en que el contexto es totalmente ajeno al modelador, una contribución de peso en el análisis del contexto del dominio el cual, como se ha indicado en el apartado anterior, es fundamental para el diseño de un sistema. Un modelo de casos de uso plantea las «historias de uso» del sistema y se centran en *qué* debe hacer el sistema sin decidir *cómo* lo hará (el diseño) (Larman, 2003). De esta forma, permite declarar los requisitos funcionales que debe contemplar este sistema (recogidos en análisis del dominio), lo cual ayuda a su vez a detectar los elementos (clases e interfaces) que van a formar parte de este diseño.

Los modelos de casos de uso se basan en la redacción –generalmente no exhaustiva– de los requisitos funcionales del sistema. Esta redacción se puede acompañar de un diagrama que muestra un conjunto de casos de uso, sus actores y sus relaciones y contribuye, entre otras cosas, a tener una imagen global del funcionamiento arquitectónicamente significativo del sistema.

Por otro lado, también se puede pensar en un caso de uso como aquél que proporciona un resultado observable de valor para un actor del sistema (Jacobson et al., 2000). La obtención de cada uno de estos elementos de valor para el usuario, implica la implementación de un mayor o menor número de acciones, cuya descripción puede ser recogida en un caso de uso antes de ser detallada en forma de colaboraciones entre clases (diseño).

En consecuencia, para capturar los requisitos del sistema, en vez de preguntarse directamente “¿cuáles son los casos de uso del sistema?, conviene comenzar preguntándose “¿qué elementos de valor debe proporcionar el sistema?” o “¿cuáles son los objetivos del sistema?”. En este sentido, el nombre del caso de uso para un objetivo de usuario debería reflejar dicho objetivo, para resaltar este punto de vista (Larman, 2003). En nuestro dominio, si uno de los objetivos del sistema es ajustar un modelo, podríamos definir el caso de uso *Ajustar Modelo*.

A continuación, se muestra el diagrama de casos de uso del *framework* MLG:


Modelador
Modelado Estadístico Guiado
Especificar Modelo
Especificar Fórmula
Especificar Nueva Fórmula
Especificar Distribución Familia Exponencial
Especificar Función Enlace
Especificar Nueva Función Enlace
Especificar Función de Variancia
Especificar Nueva Función de Variancia
Especificar Discrepancia
Especificar Nueva Distribución
Especificar Pesos iniciales
Especificar Nuevos Pesos Iniciales
Especificar Valores Ajustados Iniciales
Ajustar Modelo
Algoritmo Estimación
OLS
IWLS
Etc.
EE(b) y Sig
ICb
Discrepancia (Bondad de Ajuste)
R2
Pseudo-R2
AIC
BIC
R2 Ajustada
X2 Pearson
Cp Mallows
Etc.
Comparar Modelos
Incr. Discrepancia
Incr. R2
Incr Pseudo-R2
PRE
Etc.
Significación Estadística
F
ChiSqr
Evaluar Modelo
Residuales Ordinarios
Residuales Pearson
Residuales Discrepancia
Etc.
Medidas de influencia
Validación Cruzada
Bootstrap
Etc.
Diagnóstico Sobredispersión
Multiplicador Lagrange
LR
Prueba de Wald
Etc.
Interpretar Modelo
Valores predichos
EE Predicción
Valores ajustados
Probabilidad Predicha
Etc.
Etc.
Modelado Automático
Selección Forward
Selección Backward
Selección Stepwise
Todos los Modelos
<<include>>
<<extend>>

Tal como se ha indicado, uno de los objetivos principales del modelo de casos de uso es capturar aquellas acciones que producen un resultado de valor para el usuario. Esto es especialmente útil cuando el modelador es ajeno al campo sustantivo en el que se circunscribe el sistema a modelar. En caso contrario, los casos de uso pueden resultar un artefacto que proporciona un valor informativo general relativamente bajo. Sin embargo, aun siendo un conocedor o incluso un experto en el sistema a modelar, puede resultar aclarador el hecho de redactar ciertos casos de uso ya sea por su especial relevancia o por su particular complejidad. Atendiendo a estas consideraciones y con el objetivo adicional de ejemplificar su redactado –siguiendo la propuesta de Larman (2003)–, a continuación se muestra una selección de la redacción de casos de uso.

Caso de uso: Especificar Modelo
Actor Principal: Modelador **Escenario principal de éxito** 1. El Modelador *incluye* Especificar Fórmula. 2. El Modelador *incluye* Especificar Distribución Familia Exponencial.

Caso de uso: Especificar Fórmula
Actor Principal: Modelador **Escenario principal de éxito** 1. El Modelador especifica el tipo de fórmula (GLMFormula, GAMFormula, etc.). 2. El Sistema localiza el tipo de fórmula en el repositorio. 3. El Modelador especifica la expresión de la ecuación estructural del modelo. **Extensiones** 2a. El Sistema no encuentra en el repositorio el nombre del tipo de fórmula. 1. El Modelador incluye en el repositorio el nombre del nuevo tipo de fórmula y su especificación.

Caso de uso: Especificar Distribución Familia Exponencial

Actor Principal: Modelador

Escenario principal de éxito

1. El Modelador indica la distribución del componente aleatorio, la función de enlace y la función de variancia que definen una distribución de la familia exponencial.
2. El Sistema localiza la distribución de la familia exponencial, la función de enlace y la función de variancia en el repositorio.

Extensiones

2a. El Sistema no encuentra en el repositorio la combinación distribución, función de enlace y función de variancia.

1. El Modelador incluye en el repositorio la especificación de una nueva distribución.

Caso de uso: Aplicación del algoritmo de estimación (IWLS)

Actor Principal: Modelador

Precondiciones: Existe un modelo de la familia exponencial especificado.

Escenario principal de éxito

1. El Modelador ejecuta el proceso de ajuste especificando IWLS como algoritmo de estimación.
2. El Sistema recupera la función de enlace especificada en Especificar Función Enlace y la aplica a los valores observados.
3. El Sistema regresa el vector resultante del paso anterior sobre las variables explicativas mediante OLS.
4. El Sistema calcula los valores predichos a partir de las estimaciones obtenidas en el paso anterior.
5. El Sistema solicita a la distribución la inversa de la función de enlace, y la aplica sobre los valores obtenidos en el paso anterior para obtener los valores esperados.
6. El Sistema calcula y guarda (Z) la diferencia entre los valores esperados y observados.
7. El Sistema genera un matriz de pesos (W) cuya diagonal recoge las variancias de los valores esperados.
8. El Sistema genera una estimación por mínimos cuadrados ponderados usando W y regresando Z sobre las variables explicativas.

El Sistema repite los pasos 4 a 8 hasta que se cumpla un criterio de convergencia determinado o se llegue al número máximo de iteraciones.

9. El Modelador aplica un algoritmo de reajuste adicional en caso de sobredispersión.

Extensiones

* El Modelador ejecuta el proceso de ajuste especificando OLS como algoritmo de estimación.
* El Modelador ejecuta el proceso de ajuste especificando Backfitting como algoritmo de estimación.
* El Modelador ejecuta el proceso de ajuste especificando IWLS como algoritmo de estimación.

8. MODELO DE DISEÑO

El diagrama de casos de uso del *framework* MLG (presentado en el apartado anterior) muestra los casos de uso arquitectónicamente significativos para el posterior diseño del sistema. La intención de este diagrama es proporcionar los requisitos funcionales (de comportamiento) que debe asumir cada subsistema, esto es, especificar qué debe hacer el *framework* que se pretende diseñar. En este sentido, los casos de uso que conforman un determinado subsistema se encuentran relacionados, indicando de manera esquemática la funcionalidad que aportará ese bloque al sistema.

El diseño del *framework* se esboza en este apartado, con la intención de especificar cómo solucionar estas necesidades o requisitos de funcionamiento. Para ello, se plantea en cada subsistema una discusión sobre la selección de patrones de diseño adecuados que resuelvan dichos requisitos de funcionalidad. Una vez expuesto el problema, se analizará de manera específica qué patrón de diseño puede ser aplicable en ese contexto, evaluando en todo momento las condiciones que deben darse para que tenga sentido su aplicación –*véase* el apartado 3.2 para una descripción general del concepto de *patrón de diseño*, de sus elementos esenciales (Gamma et al., 2003).

8.1. SUBSISTEMA ESPECIFICACIÓN DEL MODELO

La especificación del modelo es la primera fase del proceso de modelado estadístico (*véase* apartado 6.1). Este subsistema se basa en la especificación de la ecuación del modelo que se pretende ajustar, además de los supuestos del componente aleatorio y la función de enlace que relacione dicho componente aleatorio con el predictor lineal.

8.1.1. La clase Formula y el patrón «Observer»

La especificación de la fórmula de un modelo define su ecuación estructural y el tipo de modelo estadístico que se quiere ajustar (GLM, GAM, etc.). En ese sentido, se requiere un objeto de tipo *Formula* que almacene información sobre esta especificación y que disponga, entre otros, de una serie de métodos para tratar las peticiones de análisis y desglose de las diferentes partes de su expresión. La clase *Formula* se extrae del caso de

uso *Especificar Fórmula* (subapartado anterior). Es una clase abstracta (figura 28) de la que derivan clases específicas (*GLMFormula, GAMFormula*, etc.), que son las encargadas de instanciar objetos fórmula a través de sus constructores, los cuales reciben como parámetro la expresión de la fórmula, y que invocan automáticamente a su método evaluate*( )*[1]. El método evaluate*( )* es un método abstracto y privado que debe ser implementado por las clases derivadas de *Formula*, con la intención de evaluar si la expresión de la fórmula es sintáctica y semánticamente correcta (es decir, si su estructura se adecua al tipo de modelo en cuestión –*GLM*, *GAM*, etc.–). Este método devuelve un valor de tipo *boolean* (*verdadero* o *falso*) indicando el resultado de esta evaluación.

Así, la creación de un objeto fórmula del tipo *GLMFormula* y con nombre *fglm* en Java se realizaría con la siguiente instrucción:

```
// Instanciación de un nuevo objeto de tipo GLMFormula
GLMFormula fglm = new GLMFormula( "y ~ x1 + x2" );
```

Los restantes métodos de la clase *Formula* se encargan de extraer de la expresión del modelo las submatrices de datos correspondientes a los diferentes términos de la ecuación. El método más básico es *terms( )*, que genera un objeto de tipo *ArrayList* con la información sobre el rol que juega en la ecuación cada uno de sus términos. Este método es utilizado como base por los restantes métodos de extracción de datos definidos en la clase *Formula*: el método *dataModel( )* devuelve un objeto de tipo *ArrayList* que contiene la submatriz de las variables originales implicadas en la fórmula más una variable identificadora de caso; el método *designMatrix( )* se encarga de generar la matriz del diseño con columnas para cada una de las variables asociadas a los diferentes parámetros del modelo (términos principales, términos de interacción, etc.); por último, el método *extractTerm( )* puede ser llamado por cualquier objeto interesado en un vector o *array* concreto de un objeto de tipo *ArrayList* que contenga datos, especificando el término de

[1] Los métodos que, como *evaluate( )*, tienen el estereotipo {leaf} son estables para las clases derivadas, esto es, no pueden ser redefinidos. Esta restricción tiene su sentido, puesto que son métodos genéricos para cualquier subclase derivada.

interés como parámetro (como se puede observar en la figura 28, los términos extraíbles están definidos como constantes estáticas –que en UML se representan en mayúsculas y subrayadas).

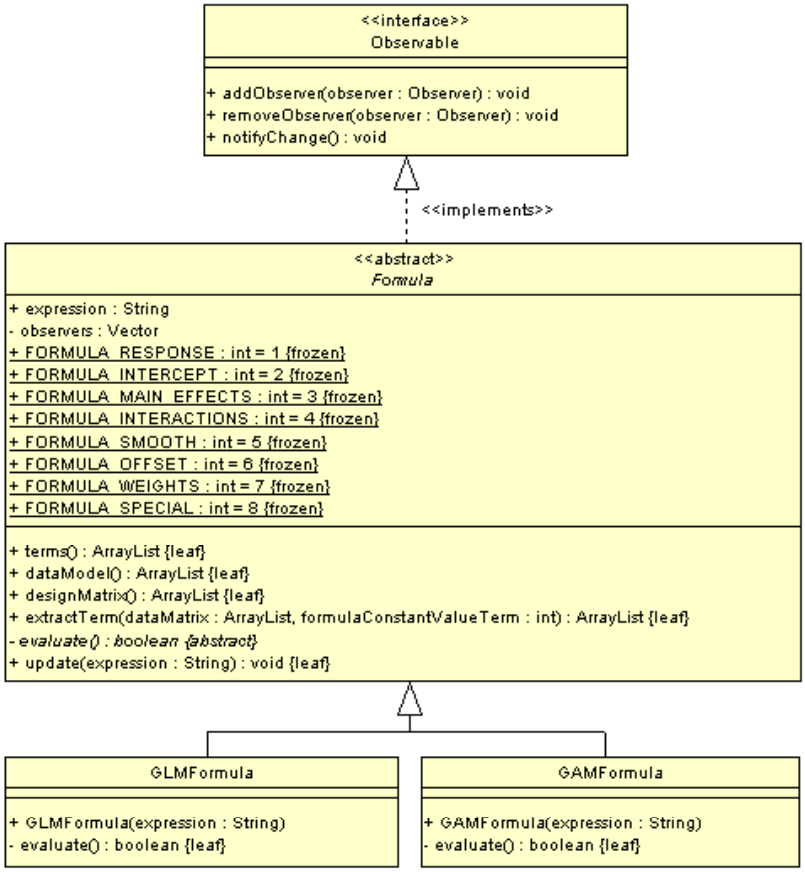


Figura 28. Clase *Formula*

Por ejemplo, la expresión Java para extraer la matriz del diseño *dm* correspondiente al objeto *fglm* instanciado anteriormente, y a continuación la variable *y* asociada al término de respuesta sería:

```
// Extracción de la matriz del diseño de un objeto de tipo GLMFormula
ArrayList dm = fglm.designMatrix( );
ArrayList y = fglm.extractTerm( dm, Formula.FORMULA_RESPONSE );
```

Para cada uno de los términos del objeto *dm* que se desee extraer se volvería a enviar el mismo mensaje al objeto *fglm*; cada término (o conjunto de términos) será recogido en un objeto *ArrayList* distinto.

Una cuestión importante que se plantea en este subsistema debido a uno de los requerimientos generales del diseño del *framework* (enunciados en el apartado 6.2) es la necesidad de que un cambio de estado en un objeto concreto sea notificado a otros objetos vinculados, de forma que estos últimos actualicen de manera automática su propio estado.

Así, por ejemplo, en nuestro *framework* si se ajusta un determinado modelo de tipo *ExponentialFamilyModel* (clase ésta que se revisa con detalle en el apartado 8.2 dedicado al subsistema de *Selección del Modelo*), interesa que el sistema vuelva a reajustar sus parámetros si se modifica la expresión de su fórmula[2]. Una posible estrategia inicial para resolver esta necesidad de comunicación entre *Formula* («fórmula» en adelante) y *ExponentialFamilyModel* («modelo» en adelante) consistiría en añadir en el objeto fórmula un atributo que contenga una referencia al objeto modelo, de modo que al efectuarse una actualización en la fórmula se realizará directamente desde ésta los cambios oportunos en el modelo. Esta estrategia presenta dos inconvenientes principales: por un lado, exige que las dos clases estén fuertemente acopladas, ya que fórmula debe conocer a modelo hasta el punto de saber qué debe cambiar de él para resolver la necesidad de sincronización entre ambos; por otro lado, si el objeto fórmula fuera utilizado por un segundo modelo debería añadirse otro atributo a fórmula para que ésta conociera a su nuevo objeto dependiente; por último, si el segundo modelo fuera de un tipo distinto al primero y requiriera un tipo de actualización distinto al de aquel, la fórmula debería conocer también esta circunstancia.

Como se desprende de lo dicho anteriormente, esta vía de solución impide desligar el conocimiento del objeto modelo por parte del objeto fórmula, y tiene un coste al añadir nuevos objetos dependientes de ésta. Frente a esta situación, el patrón de diseño *Observer* (Observador) (Stelting y Maassen, 2001; Gamma et al., 2003; Eckel, 2003) ofrece una solución general ante este tipo de demanda (figura 29). Este diseño plantea un objeto

[2] La misma necesidad de sincronización se da en nuestro *framework* entre las clases *ExponentialFamilyDistribution* y *ModelFitter* (que se revisarán más adelante) y la clase *ExponentialFamilyModel.*

observador, el cual tiene referencias a aquellos objetos observados que presentan información vinculada a su propio estado –al estilo de una hoja de cálculo–. Un objeto *observado* puede tener cualquier número de observadores dependientes de él; para añadir o quitar objetos observadores existe una interfaz que obliga a definir los métodos *addObserver(observer: Observer)* y *removeObserver(observer: Observer)*. Estos métodos gestionan la lista de objetos observadores almacenada en un atributo de la clase observada (de nombre *observers* en nuestro caso). Cada vez que el objeto observado cambia su estado, lo notifica a todos sus observadores a través del método *notifyChange( )*, también definido en la interfaz *Observable*, cuya misión es invocar al método único *actualize(observable : Observable)* de los objetos observadores, método éste definido en la interfaz *Observer* que deben implementar las clases observadoras. En respuesta, cada observador consultará al objeto observado a través de la referencia a éste último que recibe en la signatura del método *actualize( )*; de este modo los observadores pueden actualizar su estado con el del observado. La figura 29 muestra el diagrama de diseño del patrón *Observer* aplicado a las clases *Formula* y *ExponentialFamilyModel* de nuestro *framework*, y la figura 30 el diagrama de secuencia correspondiente al paso de mensajes.

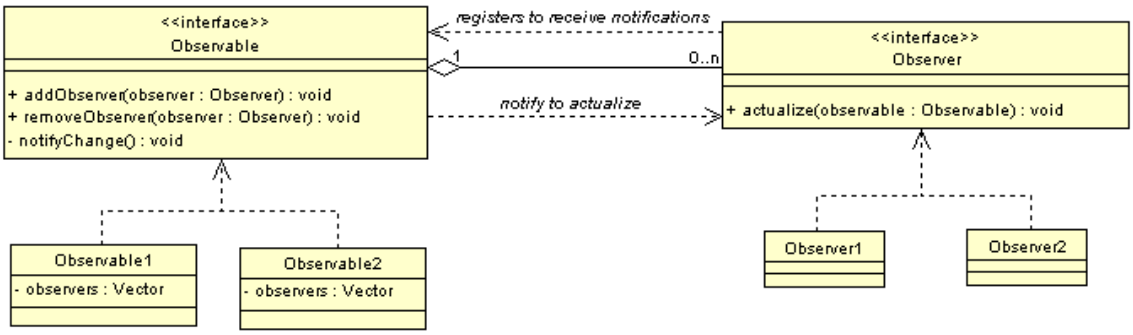


Figura 29. Diagrama de Clases genérico del patrón de diseño *Observer*

Nótese que con este patrón de diseño se evita que el objeto observado deba conocer la repercusión que pueda causar un cambio en su estado sobre el de los objetos observadores, ya que únicamente debe saber que todos los observadores responden a un mismo método *actualize( )*. En nuestro ejemplo, dado que el cambio en la expresión del objeto fórmula implica necesariamente que el objeto modelo vuelva a ajustar completamente sus

estimaciones, al invocar el método *actualize( )* de éste último sólo es necesario pasar una referencia al objeto fórmula para que el modelo la utilice en su proceso de ajuste. En este caso, la llamada desde el método *notifyChange( )* de la fórmula *fglm* al método *actualize( )* de un objeto *model* ya instanciado sería la siguiente:

```
model.actualize( fglm );
```

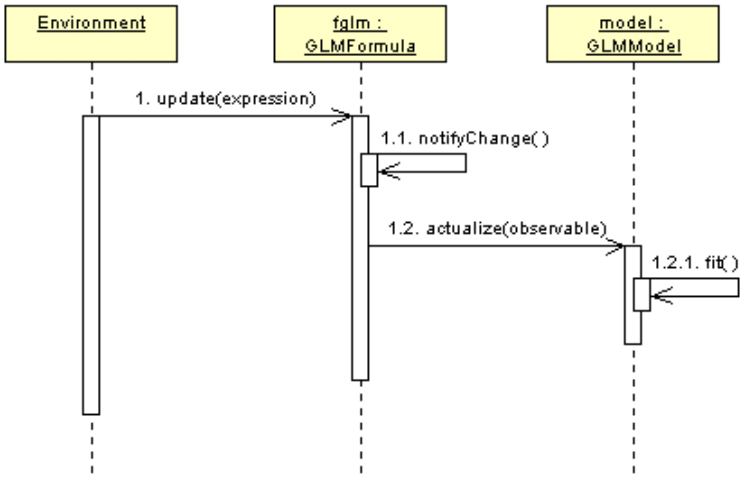


Figura 30. Diagrama de Secuencia del patrón de diseño *Observer* aplicado a la sincronización entre los objetos *Formula* y *ExponentialFamilyModel*

Si el objeto modelo tuviera que realizar actualizaciones distintas en función del tipo de cambio en el objeto fórmula, entonces sería necesario ampliar la signatura del método *actualize( )* para que incluyera algún tipo de información sobre el cambio sufrido por el objeto fórmula observado y evitar así que el modelo tuviera que "recorrer" el estado completo de la fórmula para localizar el cambio que ésta hubiera sufrido:

```
model.actualize( fglm , tipoDeCambio );
```

Por último, también relacionado con el patrón *Observer*, la clase *Formula* (figura 28) incluye el método *update( )* para efectuar la actualización de su estado (la expresión del modelo en este caso), y que en caso de invocarse realiza una llamada automática a su

método *notifyChange( )* para que aquellos objetos que dependen de la fórmula vuelvan a ejecutar sus procedimientos de actualización.

8.1.2. La clase ExponentialFamilyDistribution y el patrón «Decorator»

La especificación del modelo, además de requerir la especificación de una fórmula, como se ha visto en el subapartado anterior, también incluye la especificación de una distribución de la familia exponencial. Junto a la distribución, se determinan la función de enlace y la función de variancia empleadas en el ajuste del modelo; en general, cada distribución de la familia exponencial tiene una función de variancia fijada y, por defecto, una función de enlace específica (función de enlace canónica).

De esta manera, indicando únicamente la distribución de la familia exponencial (*Binomial*, *Poisson*, *Normal*, etc.), el sistema ajustará el modelo a partir de la variancia y función de enlace canónica asociadas a dicha distribución. Por otro lado, también se contempla la instanciación de distribuciones con un conjunto de funciones de enlace no canónicas (por ejemplo, la función de enlace logarítmica asociada a una distribución normal –clase *GaussianLog*). Por último, dadas las características especiales de la distribución *Quasi–likelihood*, se ha previsto la combinación de esta distribución con cualquier función de enlace y variancia disponibles.

Por otro lado, se ha contemplado tanto la especificación adicional de unos pesos *a priori* –empleados en el algoritmo de estimación para ponderar la matriz de pesos *W*–, requeridos por ciertas distribuciones, como la posibilidad de generar un objeto distribución que represente una versión robusta del modelo ajustado. En este sentido, un ajuste robusto requiere de unos parámetros adicionales que deben ser proporcionados al algoritmo de estimación.

Por tanto, la especificación de una distribución exponencial debe recoger todos aquellos parámetros que se consideren necesarios durante la fase de ajuste del modelo, aspecto que depende precisamente de las características particulares de dicha distribución. En este sentido, se ha considerado la inclusión en el sistema de la clase abstracta denominada *ExponentialFamilyDistribution* (figura 31), cuyas clases derivadas no abstractas instancian

un objeto distribución con funcionalidad específica para proporcionar al algoritmo de estimación aquellos cálculos que vaya solicitando durante el ajuste del modelo (aplicación de la función de enlace a los valores observados, aplicación de la inversa de la función de enlace a los valores predichos, cálculo de los residuales de discrepancia en cada iteración, etc.).

En concreto, de esta clase derivan las clases abstractas *Gaussian*, *Binomial*, *Poisson*, *Gamma*, *NegativeBinomial*, *InverseGaussian* y *Quasi*. De éstas derivan a su vez clases que implementan de manera específica los métodos abstractos de la clase base. Por ejemplo, de la clase abstracta *Binomial*, derivan las clases *BinomialLogit*, *BinomialProbit*, *BinomialCloglog* y *BinomialLog*. Cada una de estas clases permite instanciar un objeto distribución, que como se ha dicho proporciona funcionalidad al subsistema encargado del ajuste del modelo, el cual se revisa en el siguiente apartado. La figura 32 muestra esta jerarquía de distribuciones de la familia exponencial. Por otro lado, la forma en que estas distribuciones derivadas resuelven la necesidad de cálculo solicitada por el subsistema *Ajuste del Modelo* se muestra en la figura 33 (interviene el patrón de diseño *Decorator* y el acceso a métodos estáticos mediante dependencias de uso que se describen más adelante en este mismo apartado).

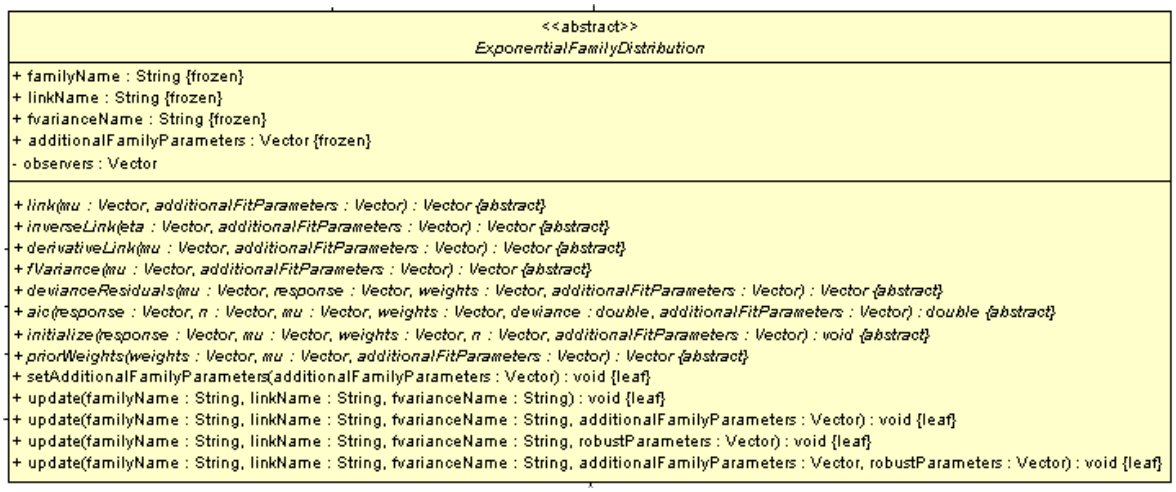


Figura 31. Clase *ExponentialFamilyDistribution*

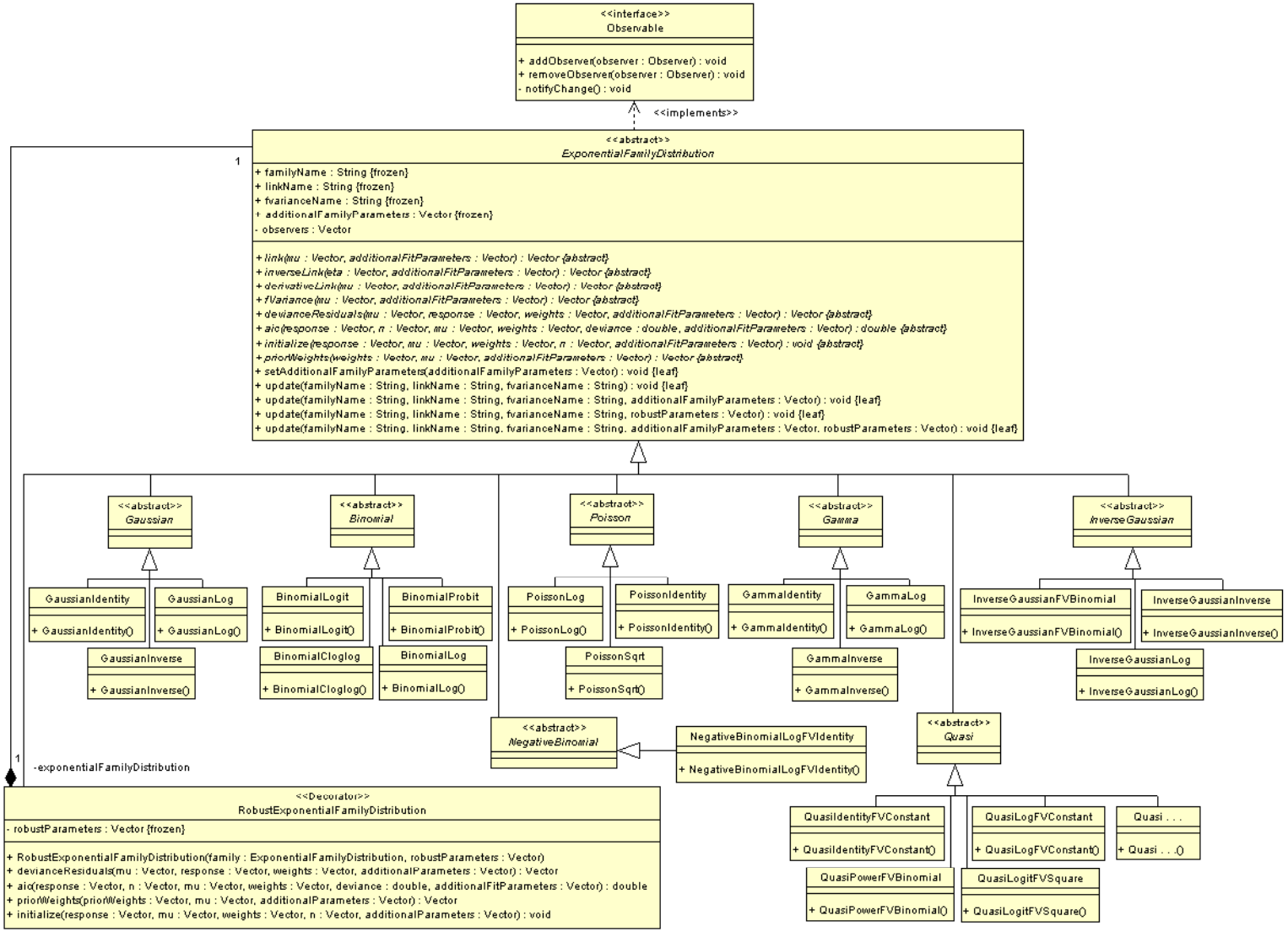


Figura 32. Jerarquía de herencia de la clase *ExponentialFamilyDistribution*

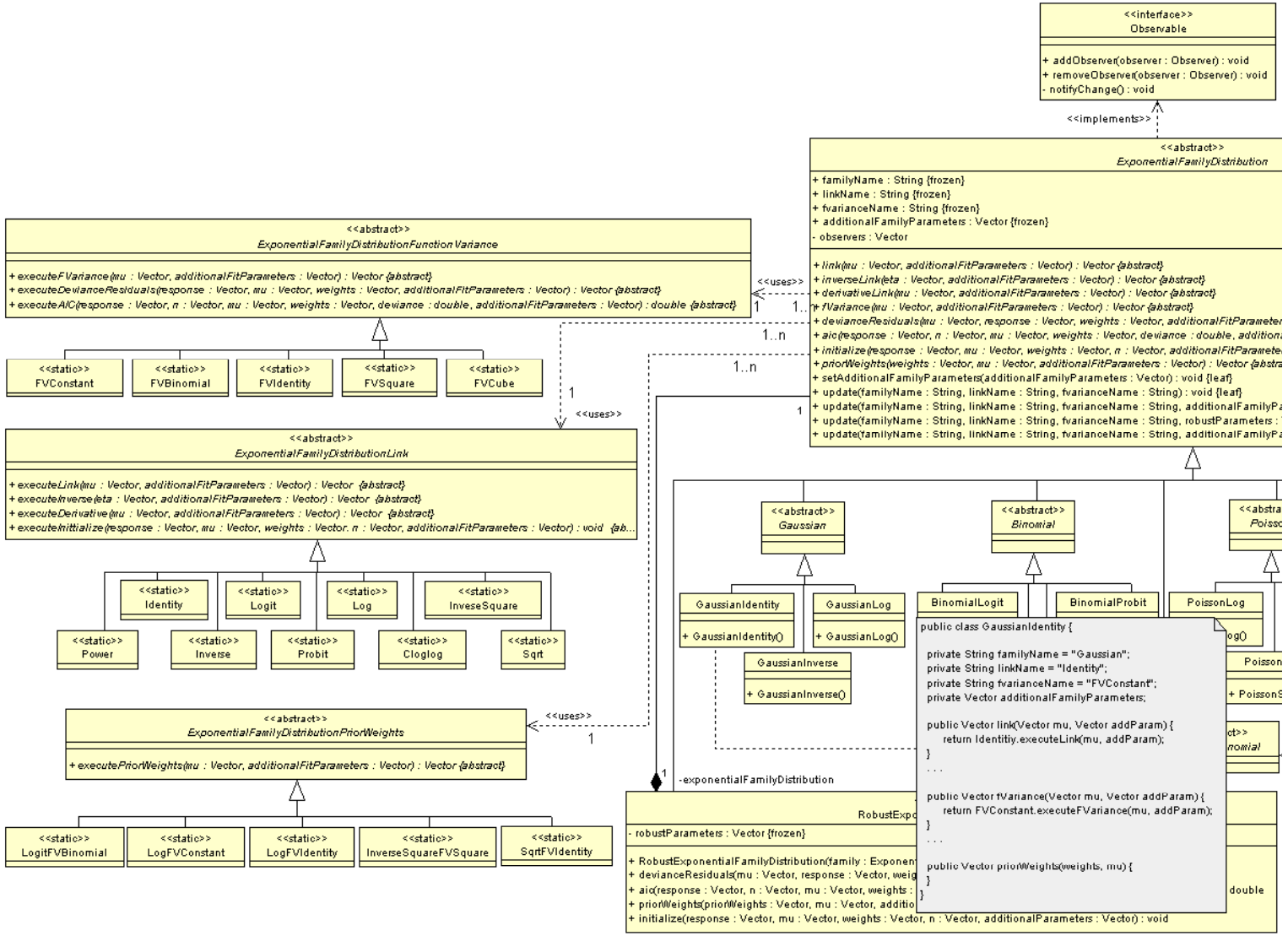


Figura 33. Relaciones de uso de los métodos de la clase *ExponentialFamilyDistribution*

Al igual que la clase *Formula* (figura 28), la clase *ExponentialFamilyDistribution* (figura 31), presenta métodos *update( )* con la finalidad de poder modificar determinados aspectos de la especificación de un objeto distribución ya creado –su familia, función de enlace, función de variancia y otros parámetros (por ejemplo, la adición de parámetros de estimación robusta)–, y controlar la notificación de estos cambios a otros objetos vinculados (patrón *Observer*). En el siguiente subsistema se muestra la vinculación de estos objetos de tipo *ExponentialFamilyDistribution* con los objetos de tipo *ExponentialFamilyModel*.

Siguiendo el paralelismo con la clase *Formula*, se observa que la jerarquía de distribuciones heredadas de *ExponentialFamilyDistribution* presentan también, por el hecho de implementar la interfaz *Observable*, los métodos *addObserver( )* y *removeObserver ( )* que permiten gestionar la lista de objetos «interesados» en los cambios de un objeto esta clase. Dichos objetos serán notificados de los posibles cambios en el estado de un objeto distribución a través del método *notifyChange( ).*

A continuación se presentan dos ejemplos de uso de la jerarquía de clases *ExponentialFamilyModel* para la instanciación de diferentes objetos distribución:

```
// Instanciación de un nuevo objeto de la clase BinomialLogit
BinomialLogit binomial = new BinomialLogit( );
```

```
// Instanciación de un nuevo objeto de la clase QuasiPowerFVBinomial
Vector lambda = new Vector( 1/3 );
QuasiPowerFVBinomial quasiPower = new QuasiPowerFVBinomial().setAdditionalFamilyParameters( lambda );
```

Las clases de la jerarquía de distribuciones, presentan una serie de dependencias de uso (figura 33) respecto a otras jerarquías de clases que proporcionan funcionalidad mediante la llamada a sus métodos estáticos. Estas clases han sido etiquetadas con el estereotipo «static» para indicar que todos sus métodos y atributos son estáticos, esto es, son accesibles a través de una llamada directa a la clase.

La ventaja de proporcionar funcionalidad mediante el acceso a métodos estáticos de una clase radica en la eficiencia de dicha estrategia; se evita tener que instanciar un objeto de la

clase para utilizar dichos métodos –permanecería en memoria sin utilidad alguna, al tratarse de un objeto función (no guarda memoria de su estado)–. En este sentido, cuando se llama a un método de una clase estática, se carga en memoria el código concreto que se desea ejecutar y es eliminado de la memoria cuando termina su ejecución (proceso automático). Por tanto, al tratarse de clases funcionales está justificado y es conveniente declarar sus métodos y atributos como elementos estáticos.

El uso por parte de un objeto distribución de los métodos estáticos de estas jerarquías de clases (*FVConstant*, *FVBinomial*, *Identity*, *Inverse*, *Probit*, *LogitFVBinomial*, etc.) (figura 33) queda delimitado en la implementación de los métodos de dicho objeto distribución (figura 31): *link( ), inverseLink( ), derivativeLink( ), fvariance( ), devianceResiduals( ), aic( ), initialize( )* y *priorWeights( )*. Un ejemplo de uso se detalla en la figura 33 (*véase* nota conectada a la clase *GaussianIdentity*).

Las clases derivadas de la clase *ExponentialFamilyDistributionFunctionVariance* proporcionan la aplicación de una implementación específica de la función de variancia, el cálculo de los residuales de discrepancia y el valor del índice AIC durante el ajuste del modelo; el objeto distribución que solicita la ejecución de estas operaciones –*executeFVariance( ), executeDevianceResiduals( )* y *executeAIC( ),* respectivamente– indica de manera unívoca en su implementación el nombre de la clase que contiene dichos métodos estáticos. Por ejemplo, un objeto de tipo *PoissonLog* solicitará la ejecución de la función de variancia a la clase *FVIdentity*, porque así se especifica en la implementación de su método *fVariance( )*, que le pasa como parámetros los elementos necesarios para su cálculo. Se utiliza la misma estrategia en la selección de las clases derivadas de *ExponentialFamilyDistributonLink* y de *ExponentialFamiliyDistributionPriorWeights*.

Otra funcionalidad recogida en el sistema es la estimación robusta de los parámetros del modelo, aspecto que se ha indicado anteriormente. Para ello, se parte de una determinada distribución de la familia exponencial cuya funcionalidad ha de ser modificada de manera apropiada, para que se adapte a los parámetros requeridos durante la ejecución del algoritmo de estimación –por ejemplo, al parámetro de escala– con el fin de robustecer la estimación de sus coeficientes. Estas modificaciones afectan, de manera fundamental, al

cálculo de la discrepancia y del AIC, al vector de pesos *a priori* y a los valores esperados iniciales en el proceso de estimación.

Para implementar esta funcionalidad, una solución directa que puede servir en general para realizar la asignación de nuevas responsabilidades a los objetos del sistema es el uso de la herencia. Siguiendo esta pauta, para efectuar la estimación robusta de las clases de tipo *ExponentialFamilyDistribution*, se podría derivar de cada una de ellas una nueva clase, y se realizarían los cambios necesarios en los métodos pertinentes –*devianceResiduals( ), aic( ), priorWeights( )* e *initialize( )*– para aportar la funcionalidad requerida en el proceso de estimación robusto. Así, por ejemplo, si se está interesado en que un objeto de la clase *PoissonLog* adquiera funcionalidad para que el algoritmo de estimación pueda realizar un ajuste robusto de un modelo, bastaría con derivar una nueva clase, *RobustPoissonLog*, y redefinir en ésta los métodos indicados. La jerarquía de clases derivadas de *ExponentialFamilyDistribution* resultante de este proceso de herencia es la que se presenta en la figura 34.

Esta manera de resolver el problema planteado es una vía factible, pero como se puede observar tiene un claro inconveniente: se ha añadido un conjunto de clases derivadas para ejercer una misma función en todas ellas, dando lugar a una explosión de clases que refleja un típico abuso de la herencia debido, en este caso, a la "mezcla" de dos grupos de funcionalidades distintas; así, las clases *Robust…* derivadas en la figura 34 son el resultado de todas las combinaciones de las distintas distribuciones de *ExponentialFamilyDistribution* y la funcionalidad añadida "robust" –común a todas ellas–. El resultado final supondrá, en definitiva, un conjunto de implementaciones redundantes (figura 34).

La solución a este tipo de situaciones en general pasa por diseñar una nueva clase para cada una de las nuevas funcionalidades, de forma que estas nuevas clases implementen los métodos de las clases existentes cuya funcionalidad se desee ampliar o modificar. En nuestro *framework* esta extensión ha sido planteada a partir del patrón de diseño denominado comúnmente *Decorator* (Decorador), que aparece asociado a la clase *RobustExponentialFamilyDistribution* (en la parte inferior izquierda de la figura 32).

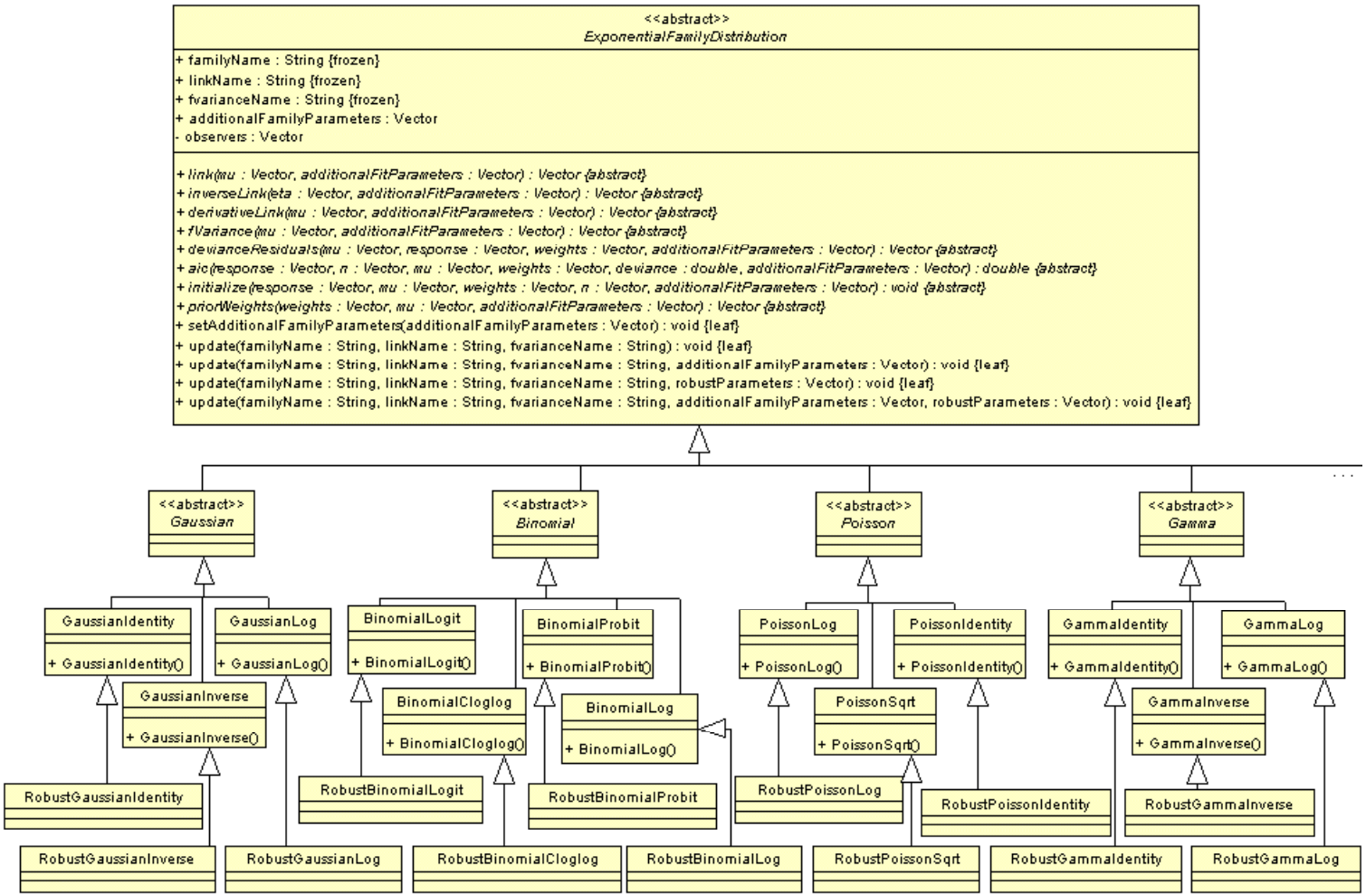


Figura 34. Uso exclusivo de la herencia para aportar funcionalidad robusta a las distribuciones de la familia exponencial

El patrón de diseño *Decorator* (Stelting y Maassen, 2001; Gamma et al., 2003; Eckel, 2003) aporta una solución adecuada a aquellos problemas que requieren de la modificación del comportamiento o funcionalidad de un objeto concreto, proporcionando una alternativa flexible a la herencia para extender dicha funcionalidad (Gamma et al., 2003). Este es nuestro caso, el objetivo principal es aumentar las atribuciones de una clase de tipo *ExponentialFamilyDistribution* para que se adapte a un proceso de estimación robusta de los parámetros de un modelo dado; en concreto, nuestra intención es modificar las capacidades internas de proceso de un objeto, sin alterar su interfaz, simplemente "decorando" esas capacidades.

En el patrón *Decorator* intervienen dos elementos clave, un objeto *decorador* y un objeto que es *decorado* por el primero; utilizando una metáfora, un objeto (el decorador) "envuelve" a otro (el decorado), de forma que el primero puede aportar funcionalidad extra a la que ofrece el objeto que ha sido decorado. A su vez, un objeto decorador puede ser decorado por otro objeto, y así sucesivamente. En este sentido, el decorador necesita compartir una misma interfaz que el objeto decorable, y debe tener como atributo interno una referencia a dicho objeto o a cualquier otro cuya clase implemente esta interfaz; además, el decorador debe implementar los métodos de la interfaz para que activen o llamen los mismos métodos del objeto referenciado en su atributo.

La figura 35 muestra una representación genérica de este patrón de diseño. Como se observa en la nota asociada a la clase *Decorator1*, ésta redefine el método *method2( )*, añadiéndole cierta capacidad de procesamiento adicional. En este sentido, un objeto de tipo *Decorator1* creado como:

```
Decorable d = new Decorator1( new Decorable2( ) );
```

se comportaría exactamente igual que un objeto de la clase *Decorable2*, excepto en su método *method2( )*, que realizaría cierto procesamiento adicional previo al realizado por el *method2( )* de *Decorable2*.

Otra de las ventajas del patrón *Decorator* respecto al uso exclusivo de la herencia, estriba en el hecho de que proporciona una manera más flexible de añadir responsabilidades a los

objetos. Con los decoradores se pueden añadir y eliminar responsabilidades en tiempo de ejecución simplemente poniéndolas y quitándolas. Por el contrario, tal como se ha comentado anteriormente, la herencia requiere crear una nueva clase para cada responsabilidad adicional (figura 34), lo que da lugar a muchas clases diferentes e incrementa la complejidad estática del sistema. Por otro lado, utilizando este patrón se pueden diseñar diferentes clases decoradoras para un determinado grupo de clases decorables permitiendo así mezclar responsabilidades.

Además, el uso de una clase que "decora" a otras permite un tratamiento exquisito de la encapsulación de objetos, puesto que el objeto decorador no necesita en absoluto conocer la funcionalidad del objeto decorado, y a su vez, el objeto decorado no sabe nada de sus decoradores, puesto que sólo le añaden un revestimiento exterior.

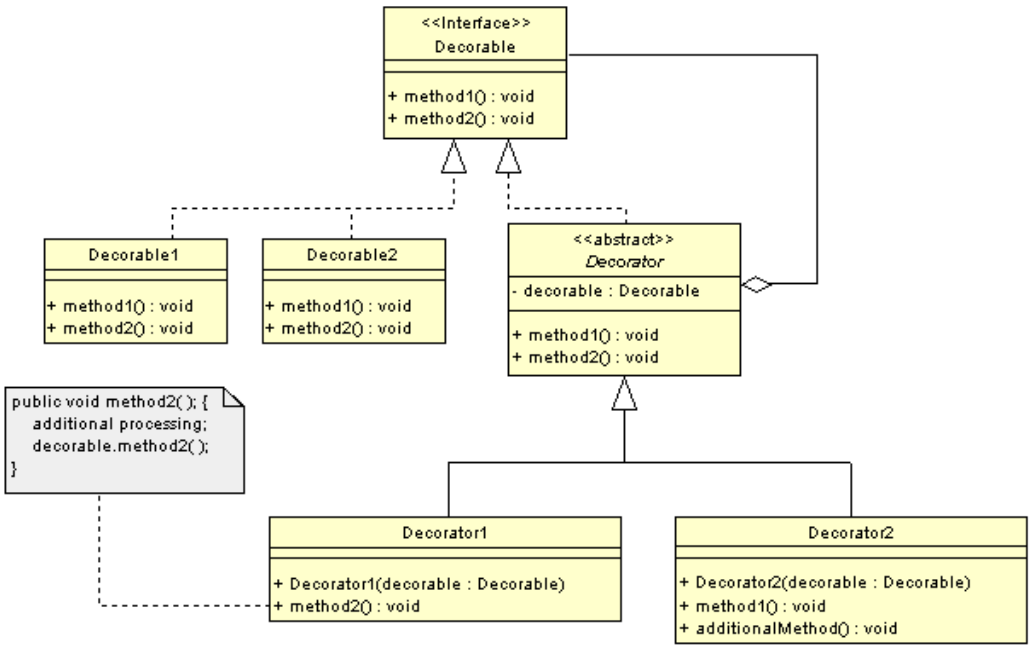


Figura 35. Estructura general del patrón de diseño *Decorator*
(Adaptado de Ocaña y Sánchez, 2003)

El patrón *Decorator* asociado a nuestro subsistema (figura 32), presenta como elementos decorables a todas las clases que derivan de *ExponentialFamilyDistribution*, mientras que

la clase decoradora *RobustExponentialFamilyDistribution* tiene la función de aportar funcionalidad extra para la estimación de modelos robustos. Como se puede observar en esta figura, se reproduce de manera particular el diseño del patrón mostrado en la figura 35. En este caso, la interfaz viene representada por una clase base abstracta. La relación que se establece entre la clase decoradora y las clases decorables es una relación de composición (variante de la agregación, representada con un rombo relleno). La composición entre objetos denota que un objeto compuesto (zona del rombo) –en este caso representa al decorador– mantiene una relación estrecha con los objetos que pueden ser decorables (objetos distribución); esta relación se realiza guardando en un atributo del decorador una referencia al objeto decorado. En nuestro caso, un objeto decorador de tipo *RobustExponentialFamilyDistribution* no puede existir sin un objeto decorable de tipo *ExponentialFamilyDistribution* al que pueda "robustecer". El siguiente ejemplo ilustra el procedimiento en sintaxis Java para la instanciación de una distribución robusta en base al patrón *Decorator* descrito:

```
// Instanciación de un nuevo objeto de la clase RobustExponentialFamilyDistribution con
// parámetro de escala = 0.5
PoissonLog poisson = new PoissonLog( );
Vector scaleParameter = new Vector( 0.5 );
RobustExponentialFamilyDistribution robustBinomial =
        new RobustExponentialFamilyDistribution( poisson, scaleParameter );
```

En la figura 36 se muestra un diagrama de secuencia asociado al patrón de diseño *Decorator*. Concretamente, se representa la ordenación temporal de mensajes entre los diferentes objetos implicados en el proceso de estimación robusta de los parámetros de un modelo, a partir de la especificación de una distribución de tipo *QuasiPowerFVBinomial* que ha sido robustecida.

Para acabar la exposición de los argumentos del diseño del subsistema *Especificación del Modelo*, es necesario realizar algunos comentarios adicionales sobre la forma de extender sus jerarquías de clases, cuya imagen global se presenta como anexo de este documento en el Diagrama de Diseño Completo de este subsistema. El investigador que haga uso del

framework puede extender tanto las dos clases base abstractas principales *Formula* y *ExponentialFamilyDistribution*, así como las clases de soporte a esta última (*ExponentialFamilyDistributonFunctionVariance*, *ExponentialFamilyDistributonLink* y *ExponentialFamiliyDistributionPriorWeights*), utilizando directamente el mecanismo de la herencia.

A continuación se presenta, a modo de ejemplo, el código necesario para definir una nueva clase derivada de *ExponentialFamilyDistribution*, en concreto la clase *NegativeBinomialLogFVIdentity* que permite ajustar un modelo de Poisson en presencia de sobredispersión[3].

```
// Definición de la distribución Binomial Negativa con función de enlace Log y variancia Identidad
public class NegativeBinomialLogFVIdentity extend ExponentialFamilyDistribution {

    // Declaración de los atributos de la clase
    public final String familyName = "Negative Binomial", linkName = "Log"; fvarianceName = "Identity";
    public double theta = null;
    private Vector observers = null;

    // Definición del método constructor de la clase (no requiere parámetros)
    NegativeBinomialLogFVIdentity( ) { }

    // Definición del método que permite asignar el parámetro de dispersión Theta
    public void setAdditionalFamilyParameters( Vector additionalFamilyParameters ) {
            this.theta = additionalFamilyParameters[ 1 ];
    }
    // Definición del método Función de Enlace
    public Vector link( Vector mu, Vector additionalFamilyParameters ) {
```

[3] Este es el procedimiento a seguir para extender mediante la herencia el resto de jerarquías del *framework*. En el siguiente apartado se describe el paso adicional necesario para que las nuevas clases definidas sean operativas desde el *framework*, que consiste en añadir la descripción y el nombre de dichas clases en unos repositorios que son finalmente utilizados internamente por parte de algunas clases especiales de tipo "constructor".

```
        return ( log( mu / ( mu + theta ) ) );
}
// Definición de la función Inversa de la Función de Enlace
public Vector inverseLink( Vector eta, Vector additionalFamilyParameters ) {
        vector eEta = exp( eta );
        return ( ( eEta *  theta ) / ( 1 – eEta ) );
}


// Definición de la función Derivada de la Función de Enlace
public Vector derivativeLink( Vector mu, Vector additionalFamilyParameters ) {
        return ( theta / ( mu * ( mu + theta ) ) );
}
// Definición de la Función de Variancia
public Vector fVariance( Vector mu, Vector additionalFamilyParameters ) {
        return ( mu * ( 1 – mu / theta ) );
}
// Definición de la función que calcula los Residuales de Discrepancia
public Vector devianceResiduals( Vector mu, Vector response, Vector weights,
                                        Vector additionalFitParameters ) {
        return ( 2 * weights * ( response * log( max( response, 1 ) / mu ) – ( response + theta ) *
                log( ( response + theta ) / ( mu + theta ) ) ) );
}
// Definición de la función que calcula el índice AIC
public Vector aic( Vector response, Vector n, Vector mu, Vector wt, double deviance,
                                        Vector additionalFitParameters ) {
        return ( 2* sum( wt / n ) * distBinomial( round( wt * y ), round( wt ), mu, “log” );
}
// Definición del método de inicialización de valores (en este caso el vector de valores ajustados)
public void initialize( Vector response, Vector mu, Vector wt, Vector n, Vector additionalFitParameters ) {
        mu = response + (  ( response == 0 ) / 6 );
}
// Definición de la función de cálculo de los Pesos a priori
public Vector priorWeights( Vector weights, Vector mu, Vector additionalFitParameters) {
        return weights;
}
```

Una vez definida la nueva clase, su uso sería:

```
Vector theta = new Vector( 1.5 );
NegativeBinomialLogFVIdentity  negbin = new NegativeBinomialLogFVIdentity( );
negbin.setAdditionalFamilyParameters( theta );
```

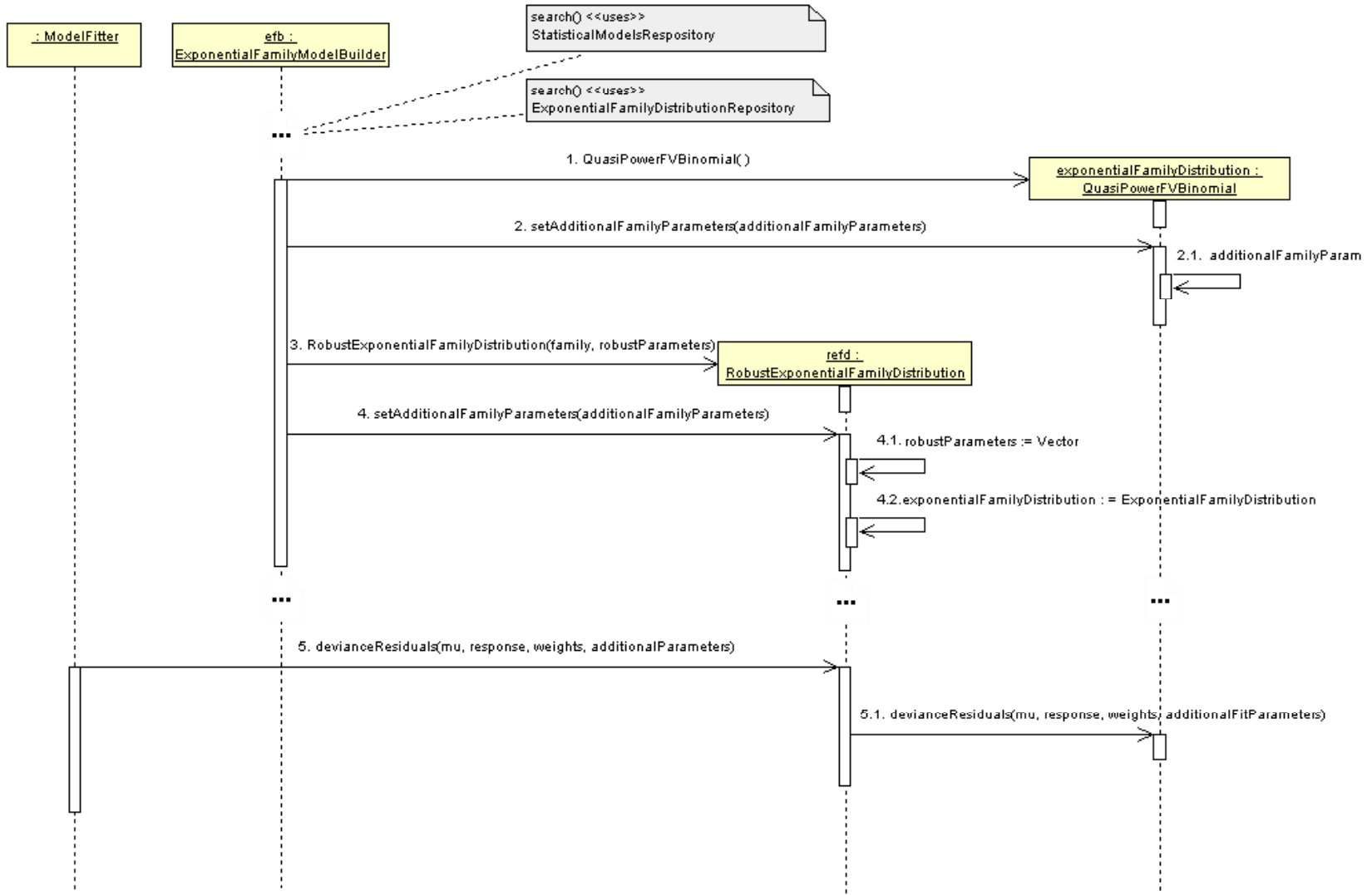


Figura 36. Diagrama de Secuencia asociado al patrón de diseño *Decorator*

8.2. SUBSISTEMA SELECCIÓN DEL MODELO

8.2.1. La clase base ExponentialFamilyModel y los patrones «Builder», «Singleton» y «Strategy»

Un aspecto importante del diseño de nuestro *framework*, consiste en centralizar los resultados de la estimación de un modelo en un objeto único, con una estructura concreta que recoja el producto del ajuste del modelo (coeficientes, valores ajustados, residuales, discrepancia, etc.), y una interfaz que gestione desde la creación del propio objeto, pasando por la delegación de responsabilidades del proceso de estimación a otros objetos y el almacenamiento de los resultados del proceso en el propio objeto.

Este planteamiento de diseño supone separar de manera explícita la estructura de un objeto de su estado, aspecto fundamental en la programación orientada a objetos; esto es, por un lado se crea el objeto, el cual hereda el comportamiento y atributos de la clase a la que pertenece, y por otro lado, este objeto va modificando su estado (los valores de sus atributos) a medida que recibe mensajes propios o de otros objetos que le instan a realizar alguna acción sobre sus datos, a partir de la interfaz de métodos que implementa.

Por otro lado, el problema de diseño que se plantea en este sentido, se centra en la necesidad de poder decidir en tiempo de ejecución qué tipo de modelo estadístico se desea ajustar (GLM, GAM, etc.). Un mal diseño de dicho problema, consistiría en integrar en una única clase la codificación de la decisión de elegir las opciones disponibles, a partir de sentencias condicionales de tipo *if*, e implementar dentro de la misma clase las particularidades asociadas a cada opción. Este planteamiento, típico de la programación estructurada, es evitable en la programación orientada a objetos (OO).

La solución que planteamos permite integrar la funcionalidad común en una única clase (la clase base), y recoger en clases derivadas la funcionalidad que varía. En nuestro sistema, esta necesidad se resuelve a través de la clase base *ExponentialFamilyModel* y las clases derivadas *GLMModel* y *GAMModel* (figura 37).

Como se puede observar, la clase *GLMModel* contiene un conjunto de atributos definidos en la clase base que permiten almacenar los principales resultados de un proceso de ajuste estadístico: *coefficients* para el vector de coeficientes estimados; *fitted* para almacenar el vector de valores ajustados m_i en la escala de la variable de respuesta; *eta* para guardar los valores ajustados en la escala del predictor lineal; *workingWeights* para recoger los pesos obtenidos tras la última iteración del algoritmo de estimación; *ordinaryResiduals* y *devianceResiduals* que contienen los residuales ordinarios (diferencia entre la respuesta y el valor ajustado por el modelo para cada caso) y los residuales de discrepancia, respectivamente; *modelDeviance*, *modelAIC*, *modelDf*, y *nullDf*, para la discrepancia y el valor del índice AIC, y los grados de libertad del modelo y del modelo nulo; *variancesCovariances* contiene la matriz de variancias y covariancias de los parámetros del modelo; *iterations*, *converged* y *boundary* guardan el número de iteraciones empleado en el proceso de estimación, si el algoritmo ha alcanzado o no el criterio de convergencia especificado, y si se han producido valores fuera del dominio aceptable durante los cálculos; finalmente, *fitStatistics* y *coefSig* representan los atributos que almacenarán todos los valores pertinentes de índices de bondad de ajuste global del modelo, así como de los valores de significación e intervalos de confianza de los parámetros estimados. Nótese que estos dos últimos atributos son del tipo *Vector*, lo que significa que están diseñados para permitir añadir un número indeterminado de índices en función de las necesidades concretas del investigador. En un apartado posterior se expone el modo en que puede realizarse esta extensión mediante los métodos *fitStatistic* y *coefSigCI* que deben implementar las clases derivadas de la clase base *ExponentialFamilyModel*.

También en la figura 37, es interesante observar como la clase *GAMModel* presenta una serie de atributos adicionales a su clase padre, particulares o específicos de este tipo de modelos (*additiveEta*, *smooth*, *variantes*, *smoothDf* y *smoothChisqr*). Esto es posible gracias al mecanismo de la herencia de clases, que evita tener que retocar el código de una clase cada vez que se desee añadir al sistema una nueva funcionalidad; basta con derivar de ésta una nueva clase y ampliar nuestras necesidades con nuevos métodos o redefinir los métodos heredados.

En cuanto a los métodos incluidos en el diseño de la clase *ExponentialFamilyModel*, además de los dos ya mencionados (*fitStatistic* y *coefSigCI*), sin duda es importante

destacar el método *fit( )*, responsable de iniciar el proceso de ajuste del modelo. Como se puede ver, este método no devuelve ningún valor (retorno de tipo *void*) y no requiere ningún parámetro para iniciar su ejecución. Tan sólo es necesario que el objeto concreto de tipo *GLMModel* o *GAMModel* haya sido instanciado. Esto es así porque, tal como se describirá más adelante, estas clases delegan toda la responsabilidad del proceso de datos y del ajuste del modelo en objetos de las clases *Formula*, *ExponentialFamilyModel* y *ModelFitter* (esta última se trata en el siguiente apartado), cuyas referencias quedan almacenadas en los atributos *formula*, *distribution* y *modelFitter*, respectivamente (*véase* la relación de agregación, simbolizada en UML con rombos vacíos conectados a la izquierda del recuadro de la clase *ExponentialFamilyModel* en la figura 37). En este sentido, la clase base incluye también el método sobrecargado *update( )*, para aportar la posibilidad de referenciar objetos *Formula*, *ExponentialFamilyDistribution* y *ModelFitter* distintos a los reseñados en la signatura del constructor de las clases *GLMModel* o *GAMModel* en el momento de su instanciación. En el caso de que el objeto modelo ya hubiera sido ajustado en el momento de ejecutar su método *update( )*, éste realiza una llamada automática al método *fit( )* para iniciar un nuevo proceso de ajuste en base al nuevo estado.

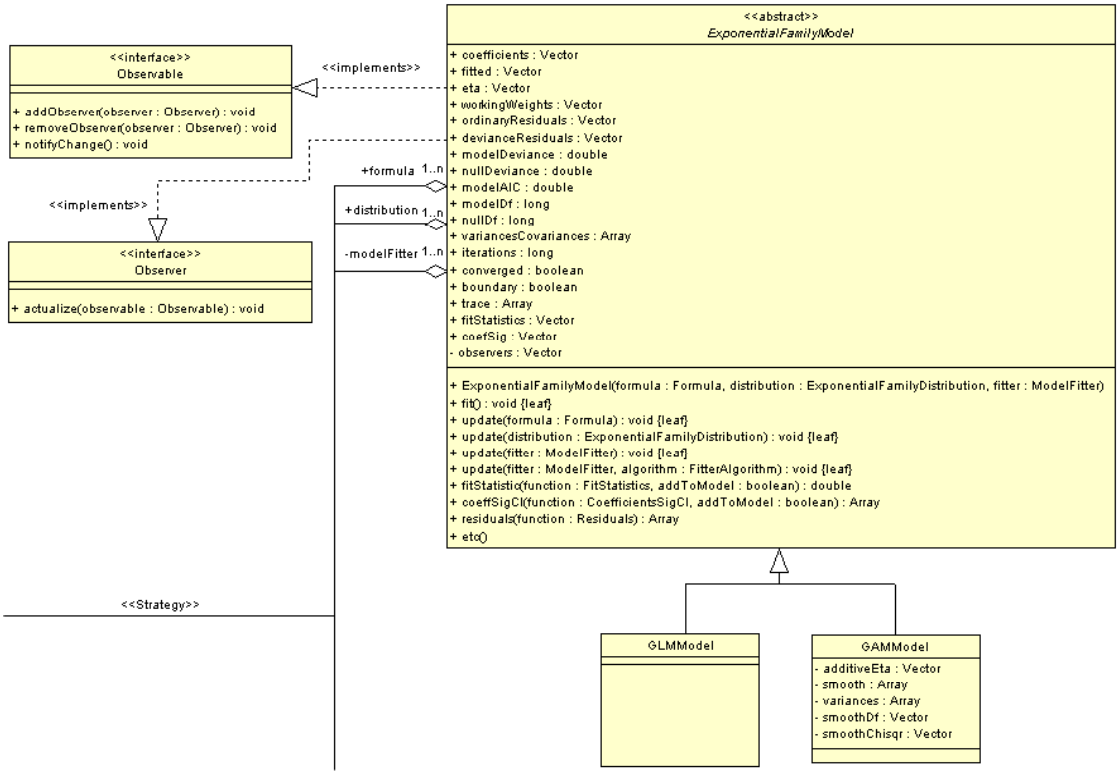


Figura 37. Clase *ExponentialFamilyModel* y derivadas

Es interesante recordar en este punto el funcionamiento del patrón *Observer* descrito en el apartado anterior. Como ya se comentó, el diseño basado en dicho patrón (*véase* las figuras 29 y 30 del subsistema *Especificación del Modelo*) permite que un cambio en los objetos que componen un *ExponentialFamilyModel* sea automáticamente notificado a éste (mediante la invocación de su método *actualize( )*) definido en la interfaz *Observer* que implementa –*véase* la parte superior izquierda de la figura 37–, para que inicie de nuevo su proceso de ajuste si lo considera necesario. Aunque no se incidirá de nuevo sobre esta funcionalidad, es la clase constructora *ExponentialFamilyModelBuilder* cuyo diseño se describe a continuación en este mismo apartado, la encargada de añadir el objeto modelo a la lista de observadores de los objetos *Formula*, *ExponentialFamilyDistribution* y *ModelFitter* durante el proceso de creación de un *GLMModel* o *GAMModel*. Por tanto, ya desde el momento de su instanciación, los objetos modelo se constituyen como observadores permanentes de los cambios en los objetos que lo componen, a no ser que el investigador decida "desacoplar" dichos objetos mediante el método *removeObserver( )* definido en la interfaz de los tres objetos *Observables* mencionados.

La aplicación del patrón *Observador* en la clase *ExponentialFamilyModel* presenta una peculiaridad remarcable: el hecho de que dicha clase es observadora (tal como se ha indicado anteriormente) a la vez que observable. De esta forma, la clase *ExponentialFamilyModel* implementa una interfaz *Observer* y una interfaz *Observable*: la primera define los métodos necesarios para ser notificada de cualquier cambio de estado de los objetos observados, y mediante la segunda, cualquier objeto que herede de la clase *ExponentialFamilyModel* (GLMModel o GAMModel) puede notificar a los objetos observadores los cambios sufridos. Tal como se detallará en apartados posteriores, en nuestro diseño existe una sola clase observadora de *ExponentialFamilyModel*: la clase *ModelComparisonFactory*.

Como se indicó anteriormente, el método *update( )* incluido en las clases derivadas de *ExponentialFamilyModel* permite realizar también un proceso de re-*fit( )* al cambiar la referencia a los objetos *Formula, ExponentialFamilyDistribution* y *ModelFitter* agregados al modelo. Es importante destacar la diferencia entre estas dos estrategias de actualización del estado y/o el comportamiento de los objetos modelo. Por un lado, el patrón *Observer* está diseñado para informar a todos los modelos que estén haciendo uso de un mismo

objeto del tipo de los tres reseñados, de un cambio en los mismos, mientras que el método *update( )* de los objetos modelo permiten cambiar el estado y/o comportamiento de los modelos asignándoles nuevos objetos agregados, sin que esto desencadene ninguna secuencia de solicitudes de actualización a otros objetos, ya que el cambio se realiza en este caso directamente en el objeto *Observer* y no en los objetos *Observables*.

Pasando a otra cuestión relevante en el diseño de las clases *ExponentialFamilyModel*, una restricción que se impone es que no se pueden crear instancias de la clase base (definida como clase «abstracta»). Es una decisión de diseño obvia, si tenemos en cuenta que la única justificación de esta clase es la de servir de interfaz para sus clases derivadas, que son las que realmente van a iniciar el proceso de modelado y quienes van a recoger el resultado de dicho proceso. En este sentido, el tipo de modelo elegido (GLM, GAM), impondrá sus particularidades en el proceso de estimación, delegando en los objetos que agrega la responsabilidades asignadas. Esta delegación se realiza a partir del patrón de diseño *Strategy* (Estrategia), en el que nos detenemos más adelante.

a. La clase ExponentialFamilyModelBuilder y el patrón «Builder»

Como se anunció en la introducción de este trabajo, uno de los objetivos de este *framework* es precisamente facilitar al usuario no experto la interacción con el sistema diseñado –esto es, que con un mínimo de instrucciones sea capaz de ajustar un modelo–. Por otra parte, de cara a cubrir objetivos de investigación, el sistema también estará preparado para hacer un uso eficiente del mismo, dirigido por el investigador.

Precisamente, para cubrir la facilidad de uso del *framework*, se ha considerado la inclusión en nuestro diseño de una clase que facilita la construcción directa de un objeto modelo, nos referimos a la clase *ExponentialFamilyModelBuilder* (figura 38). A través de un objeto de esta clase, se indican los parámetros de especificación del objeto modelo de interés y se llama a su método *getModel( )*. Esta clase representa el patrón de diseño *Builder* (Constructor), referenciado en Stelting y Maassen (2001), Gamma et al. (2003) y Eckel (2003).

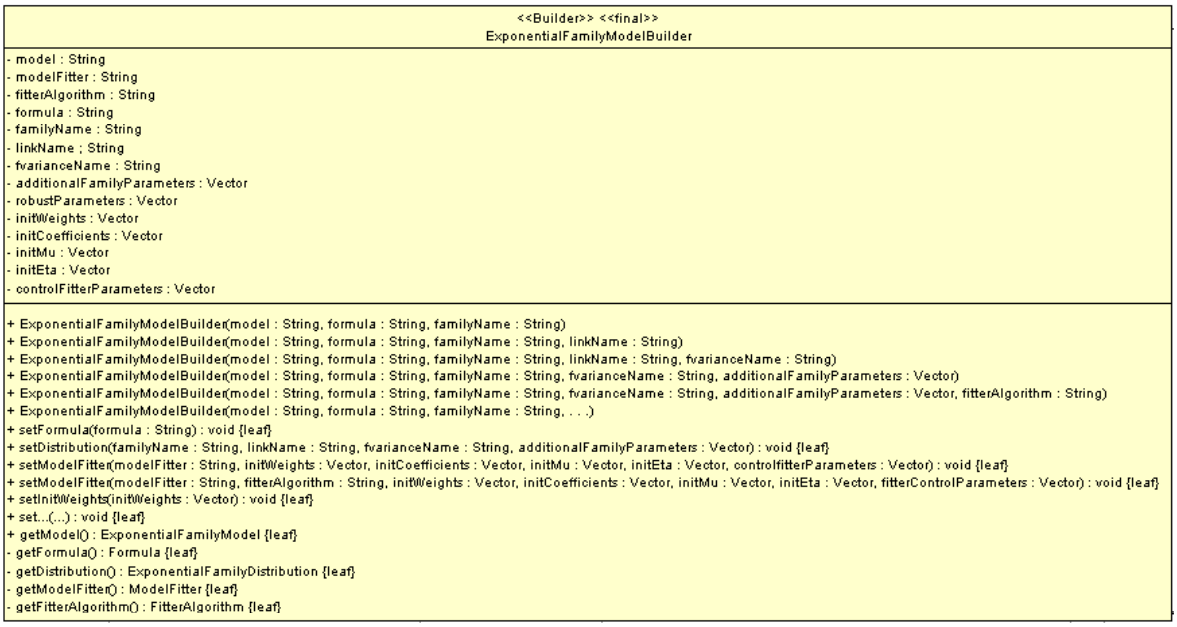


Figura 38. Clase *ExponentialFamilyModelBuilder*

En relación al patrón *Builder*, su particularidad reside en la obtención como producto final de un único objeto compuesto –de tipo *ExponentialFamilyModel*–, como resultado de la unión de las distintas partes (objetos) que lo componen. Estas partes también son instanciadas por el objeto *Builder*, aunque los métodos que ordenan dicha construcción son privados de *Builder*, y únicamente es público el método que devuelve el objeto compuesto (figura 38).

Un patrón de construcción relacionado con *Builder* es el patrón *Abstract Factory* (Stelting y Maassen, 2001; Gamma et al., 2003; Eckel, 2003). Su función es la de crear familias de objetos relacionados, pero a diferencia del primero, no los ensambla. En este sentido, los constructores de cada uno de estos objetos son públicos, puesto que no son considerados objetos «parte» –cuya lógica de creación se encapsula en el patrón *Builder*– de un objeto compuesto.

Como se observa en la figura 39, el patrón *Builder* permite obtener diferentes representaciones del objeto que se está construyendo, gracias a que los métodos de construcción de las diferentes partes del objeto compuesto pueden tener diferentes implementaciones en cada uno de los constructores concretos que derivan del constructor abstracto. Esto es posible debido a que cada uno de estos constructores concretos comparte

una misma interfaz. En este sentido, llamando al mismo método *getProduct( )* en un constructor y en otro, se pueden obtener productos distintos, aunque no sólo porque los distintos métodos de construcción de cada parte puedan tener una implementación distinta, sino también porque puede variar la manera de ensamblar esas partes. Esto aporta un control más fino sobre el proceso de construcción y, por tanto, sobre la estructura interna del producto resultante.

En nuestro caso, el objetivo del patrón *Builder* es generar un objeto modelo (de tipo *GLMModel* o *GAMModel*) que está compuesto de cuatro objetos: un objeto *Formula*, un objeto *ExponentialFamilyDistribution*, un objeto *ModelFitter* y un objeto *FitterAlgorithm*. En la figura 38, se muestran los métodos de construcción asociados a estos objetos: *getFormula( ), getDistribution( ), getModelFitter( )* y *getFitterAlgorithm( ),* de manera respectiva. Son constructores privados, encapsulados dentro del método *getModel( )* –que arranca el proceso de construcción–, al que llama el usuario para obtener directamente, y de manera transparente, un objeto modelo.

Este proceso de generación de un modelo a partir de la instanciación de sus partes queda reflejado en el diagrama de secuencia de la figura 41 (siguiente subapartado).

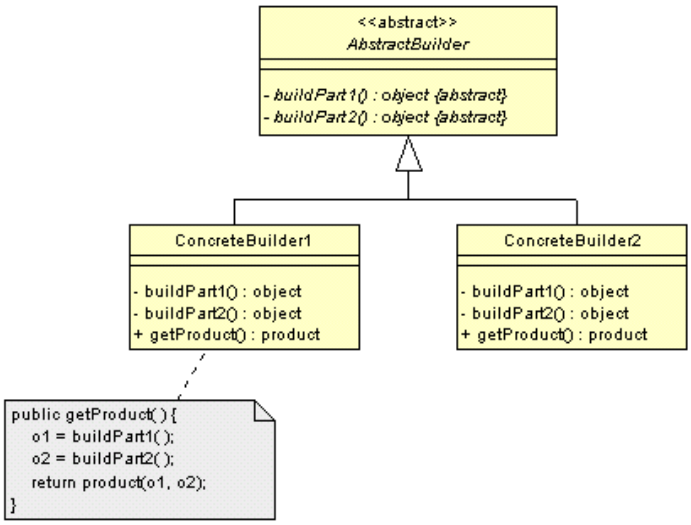


Figura 39. Patrón de diseño *Builder*

Por tanto, la utilidad del patrón *Builder* está justificada de cara a facilitar al usuario la obtención de un modelo ajustado, sin que éste se preocupe de la lógica de construcción de dicho objeto. Es un controlador entre la interfaz de usuario y el sistema.

Como anexo de este documento, se adjunta el diagrama de clases de diseño (DCD) del subsistema *Selección del Modelo*. En este DCD, se observa la responsabilidad que asume la clase *ExponentialFamilyModelBuilder* en la construcción de los diferentes productos que compondrán el objeto modelo.

Sin embargo, nuestro *framework* también permite que un usuario avanzado pueda crear de manera independiente cada objeto implicado en el ajuste de un modelo. Por ejemplo, puede instanciar un objeto de tipo *GLMModel* (figura 37), y tal como ha sido descrito, a través de sus métodos *update( )* modificar la especificación de un objeto fórmula, de un objeto distribución o de un objeto "ajustador" para reajustar el objeto modelo. En este sentido, la lógica de construcción puede ser dirigida por el investigador, proporcionando una vía de análisis de problemas centrados en la simulación. Por tanto, en realidad, no es necesario el patrón *Builder* para que funcione el sistema, aunque es muy útil para encapsular la construcción de un modelo de cara a un usuario no experto.

Otra flexibilidad asociada a la clase *ExponentialFamilyModelBuilder* (figura 38), es que dispone de una serie de constructores sobrecargados que permiten crear un objeto con información concreta sobre las especificaciones de un modelo. En función de los parámetros indicados en estos métodos constructores sobrecargados (con mismo nombre que la clase, pero con signaturas distintas), el objeto almacenará en sus atributos (de tipo *String*) unas u otras especificaciones. En concreto, una vez instanciado el objeto *Builder* con un constructor determinado, la llamada al método *getModel( )* instanciará los objetos de la clases que recogen la lógica de creación del modelo –*Formula*, *ExponentialFamilyDistribution*, *ModelFitter* y *ExponentialFamilyModel*–, pasándoles como parámetros los almacenados en los atributos de *Builder* ya mencionados.

El uso de constructores sobrecargados (y de métodos sobrecargados, en general), ofrece una comodidad y flexibilidad extra al sistema, puesto que permite ejecutar un método concreto de múltiples formas, en función de los parámetros especificados en la signatura del método. Gracias a ello, el usuario podrá decidir el número y tipo de parámetros

relacionados con la especificación del modelo, lógicamente siempre dentro de las restricciones impuestas en el diseño. En nuestro caso, se recogen en la clase los constructores más significativos (aunque se podrían incluir otras combinaciones).

Un posible problema en el diseño de esta clase podría venir dado por el hecho de que un usuario determinado decida derivar de ésta una nueva clase, con la intención de realizar cambios de implementación en la clase derivada. Si se tiene en cuenta que la clase *ExponentialFamilyModelBuilder* recoge una determinada lógica de construcción del modelo, que encapsula totalmente el proceso de ajuste, puede ser conveniente impedir esta posibilidad. Con esa intención, la clase ha sido etiquetada con el estereotipo «final», que indica que no es posible derivar de ella otra clase. De esta manera, por tanto, se asegura que el procedimiento seguirá estando bien automatizado para usuarios no expertos, que simplemente tendrán que especificar las características del modelo a ajustar –esto es, llamar al constructor de la clase *ExponentialFamilyModelBuilder*, indicando estas características– y una vez que ha sido instanciado, ejecutar su método *getModel( )*.

Otra utilidad asociada a esta clase es la posibilidad de utilizar sus métodos *set( )*, –*setDistribution( )*, *setFormula( )*, *setModelFitter( )*, etc.–, para modificar los valores de los atributos almacenados en un objeto *Builder* concreto; ello permite volver a llamar a su método *getModel( )* desde el mismo objeto para generar otros modelos con distintas especificaciones.

b. Las clases Repository y el patrón «Singleton»

Un requisito ligado a la construcción de un objeto *Builder*, pasa por controlar que los parámetros recibidos a través del constructor sean consistentes (en función de la lógica de ajuste), por ejemplo, que sea correcta la especificación de la distribución de la familia exponencial con una determinada función de enlace. Una posible solución a esta necesidad, podría consistir en definir un conjunto de sentencias condicionales *if* dentro de la clase *ExponentialFamilyModelBuilder*, una sentencia para cada combinación de clases (tipo de especificación) asociadas al ajuste del modelo. Sin embargo, el inconveniente que presenta esta opción, además de complicar el código de dicha clase, es la poca flexibilidad que aporta, puesto que si es necesario añadir una nueva entrada como resultado de la extensión

de sistema con otro tipo de distribución u otro tipo de algoritmo de ajuste, habría que modificar el código de dicha clase –introducir una nueva sentencia *if* que contemple los nuevos tipos de elementos previstos en el proceso de modelado–.

Existe una solución mucho más elegante, que consiste en el uso de repositorios del sistema (clases de tipo *ArrayList*) (figura 40), a los que accede el objeto *Builder* para verificar si la información proporcionada por el usuario es correcta. Concretamente, se han considerado dos repositorios (tablas), que recogen las distintas especificaciones posibles. Cada repositorio es una clase del sistema que contiene un campo para cada uno de los elementos de decisión en el proceso de ajuste; en concreto, almacena en cada campo la cadena de caracteres que ha de coincidir con la información almacenada en un atributo concreto de la clase *ExponentialFamilyModelBuilder* –los parámetros especificados en el constructor se almacenan en los atributos del objeto–. Si el usuario no indica alguno de estos atributos (es decir, no tiene asignado ningún valor), el objeto *Builder* busca el nombre de estos atributos directamente en el repositorio correspondiente, utilizando como clave los elementos que han sido especificados.

Así, por ejemplo, la sentencia siguiente crearía directamente tres objetos *GLMModel* utilizando el constructor más sencillo de la clase *ExponentialFamilyModelBuilder*:

```
ExponentialFamilyModelBuilder glmBuilder =
        new ExponentialFamilyModelBuilder( “GLM”, “y ~ x1 + x2”, “Binomial” );
GLMModel binomialX1X2 = glmBuilder.getModel( );
GLMModel binomialX1 = glmBuilder.setFormula( “y ~ x1” ).getModel( );
GLMModel glmQuasiX1 = glmBuilder.setDistribution( “Quasi”, “Power”, “Binomial”, new Vector( 1/2 ) ).getModel( );
```

En este caso, el objeto *glmBuilder* localizaría en el repositorio *StatisticalModelsRepository* –con el que guarda una relación de uso (*véase* la figura 40)– el registro correspondiente al tipo de modelo (“GLM”), y en el repositorio *ExponentialFamilyDistributionsRepository* la fila correspondiente a cada una de las distribuciones (“Binomial” y “Quasi”); en el primero y segundo de los modelos de tipo *GLMModel* que se solicitan a *glmBuilder* (*binomialX1X2 y binomialX1*), puesto que se ha indicado únicamente el nombre de la distribución, el constructor selecciona automáticamente en el repositorio la fila correspondiente a la

función de enlace Logit por tratarse de la canónica para la Binomial –y variancia binomial $m_i \times (1 - m_i)$.

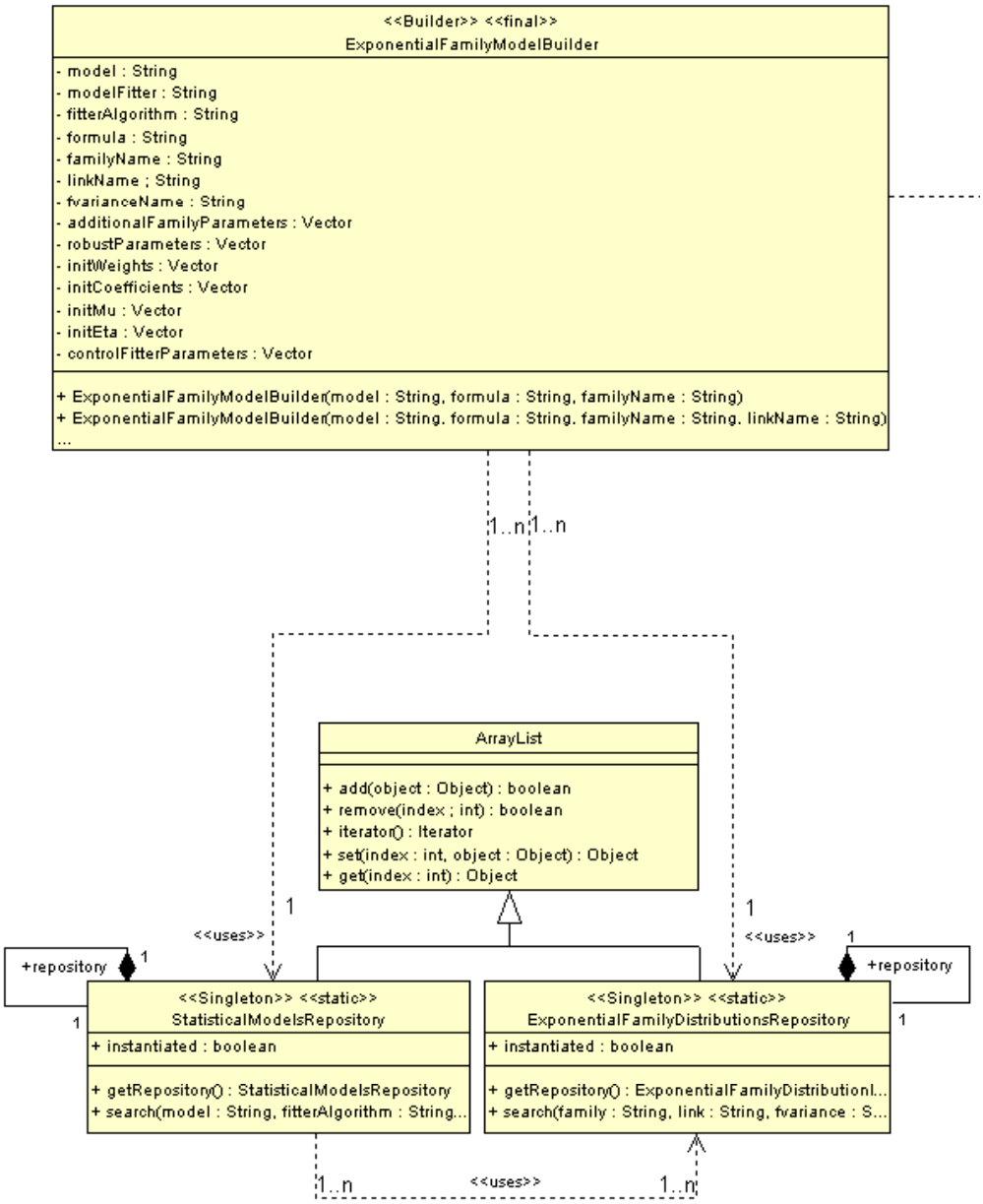


Figura 40. Repositorios del *framework* relacionados con la clase *ExponentialFamilyModelBuilder*

En estos dos repositorios del sistema se encuentran, asociadas a las etiquetas que definen el modelo desde el punto de vista del usuario, los nombres específicos de las clases que

finalmente el objeto *Builder* tiene que instanciar. Así, en el ejemplo planteado se encontraría en el repositorio *StatisticalModelsRepository* asociado a la etiqueta "GLM" el nombre de la clase *Formula* relacionada con este tipo de modelo concreto: *GLMFormula*, así como el nombre de las dos clases que proporcionan objetos para llevar a cabo el proceso de ajuste: *GLMFitter* y *IWLS* –se abordarán estas clases en el subsistema *Ajustar Modelo*–; por su parte, en el repositorio *ExponentialFamilyDistributionsRepository* se encontraría asociada a la etiqueta "Binomial" el nombre de la clase concreta a instanciar para su función de enlace canónica: *BinomialLogit*; por último, en el primer repositorio también se halla el nombre del objeto modelo compuesto por todos los objetos citados: *GLMModel*. La forma en que *Builder* resuelve internamente la creación de los objetos a partir de los nombres de sus clases podría utilizar en Java un mecanismo de "reflexión" del tipo:

```
ExponentialFamilyDistribution distribution =
        (ExponentialFamilyDistribution) Class.forName( "BinomialLogit" ).newInstance( );
```

Las entradas del repositorio, lógicamente, deben estar almacenadas en disco para cargarlas en memoria cada vez que se crea un objeto *Builder*, y de manera síncrona, actualizar este fichero con las modificaciones realizadas en dicho repositorio. Se trata de un fichero interno del *framework*, de acceso restringido.

Es importante hacer notar que el uso de estos repositorios evita tener que definir nuevas clases concretas derivadas de *ExponentialFamilyModelBuilder* para cada nuevo tipo de modelo estadístico que se decida añadir al *framework*, siguiendo el modo en que se presenta en el DCD genérico del patrón *Builder* (figura 39). Esta es una solución de diseño más óptima que la que plantea el propio patrón puesto que, aunque exista una única clase *final* de *ExponentialFamilyModelBuilder*, el investigador puede añadir funcionalidad al *framework* definiendo nuevas clases sin tener que acceder a la implementación del *Builder*, ya que como se ha comentado antes, éste resuelve mediante los repositorios la decisión de los objetos asociados al modelo sin tener que utilizar sentencias condicionales (utilizando el mecanismo de "reflexión" comentado).

Así, por ejemplo, para informar a *ExponentialFamilyModelBuilder* sobre la nueva distribución *NegativeBinomialLogFVIdentity* cuya implementación se presentó al final del apartado anterior, se utilizaría una sentencia del tipo:

```
ExponentialFamilyDistributionRepository.add( “NegativeBinomial”, “Log”, “Identity”, true,
        “NegativeBinomialLogFVIdentity” );
```

Como se puede observar, se accede directamente al repositorio de distribuciones de probabilidad de la familia exponencial mediante su método *add(*), indicando como parámetros el nombre genérico de la distribución, el de su función de enlace, el de su función de variancia, un parámetro *boolean* que indica si se trata de la función de enlace canónica para esta distribución, y finalmente el nombre real de la nueva clase que implementa la distribución añadida. A partir de este momento, los objetos de tipo *ExponentialFamilyModelBuilder* serán capaces de generar modelos del tipo *GLMModel* y *GAMModel* que contengan referencias a esta nueva clase distribución:

```
GLMModel glmNegBin =
        new ExponentialFamilyModelBuilder( “GLM”, “y ~ x1 * x2”, “NegativeBinomial” ).getModel( );
```

En la figura 41 se muestra el diagrama de secuencia del patrón *Builder*, en el que se observa la colaboración de un objeto de tipo *ExponentialFamilyModelBuilder* con los repositorios en el proceso de construcción de un objeto modelo.

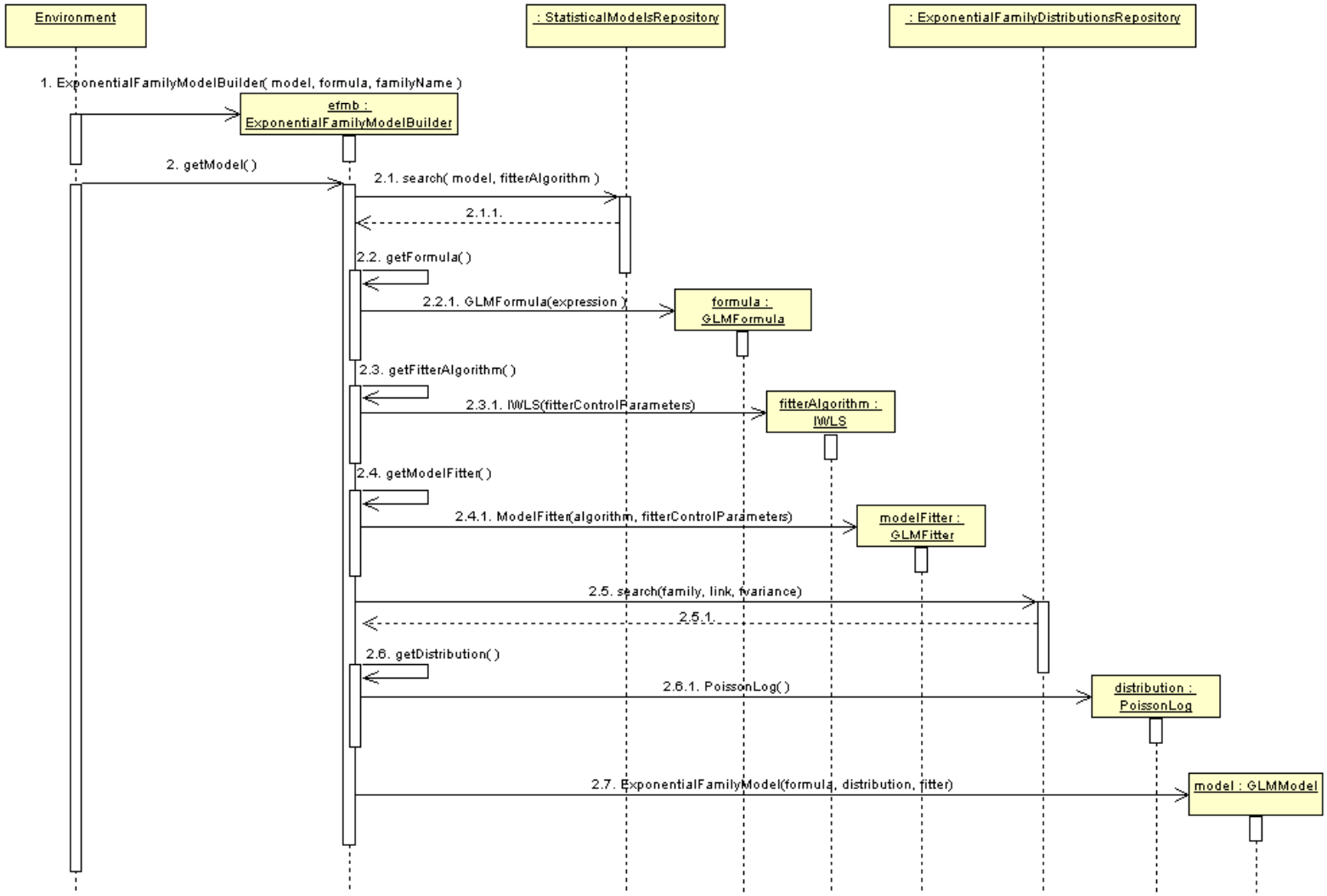


Figura 41. Diagrama de Secuencia del patrón de diseño *Builder* aplicado a la clase *ExponentialFamilyModelBuilder*

Una particularidad adicional de las clases repositorio de nuestro sistema es que su diseño se basa en el patrón *Singleton* (Único), esto es, son clases de las que puede haber una única instancia –este patrón se encuentra descrito en Gamma et al. (2003) y Eckel (2003)–. Se justifica esta decisión de diseño si se tiene en cuenta que debe existir un punto de acceso global a la información almacenada en estas tablas. Una representación del patrón *Singleton* se muestra en la figura 42.

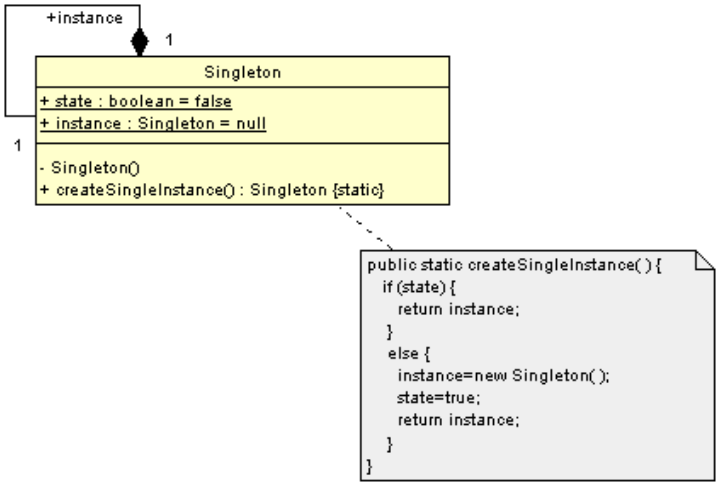


Figura 42. El patrón de diseño *Singleton*

Como se puede observar, la estructura de este patrón es la misma que la presentada en la figura 40 para las clases repositorio. En concreto, estas clases utilizan el método *getRepository( )* para crear la instancia del repositorio, y el atributo *instantiated* para controlar si el objeto ya ha sido creado. La primera vez que se llame al método *getRepository( )*, haciendo referencia directamente a la clase que lo define –método estático–, éste comprobará que el atributo *instantiated* es igual a *false* y creará el objeto repositorio; una vez ha sido creado, modificará el estado de *instantiated* a *true* evitando que durante la sesión se pueda volver a instanciar otro objeto de la misma clase.

La figura 43 muestra precisamente un diagrama de secuencia que refleja el papel del patrón de diseño *Singleton* en la instanciación de un único objeto de la clase *StatisticalModelsRepository*. El proceso de creación sería el mismo para la clase *ExponentialFamilyDistributionsRepository*.

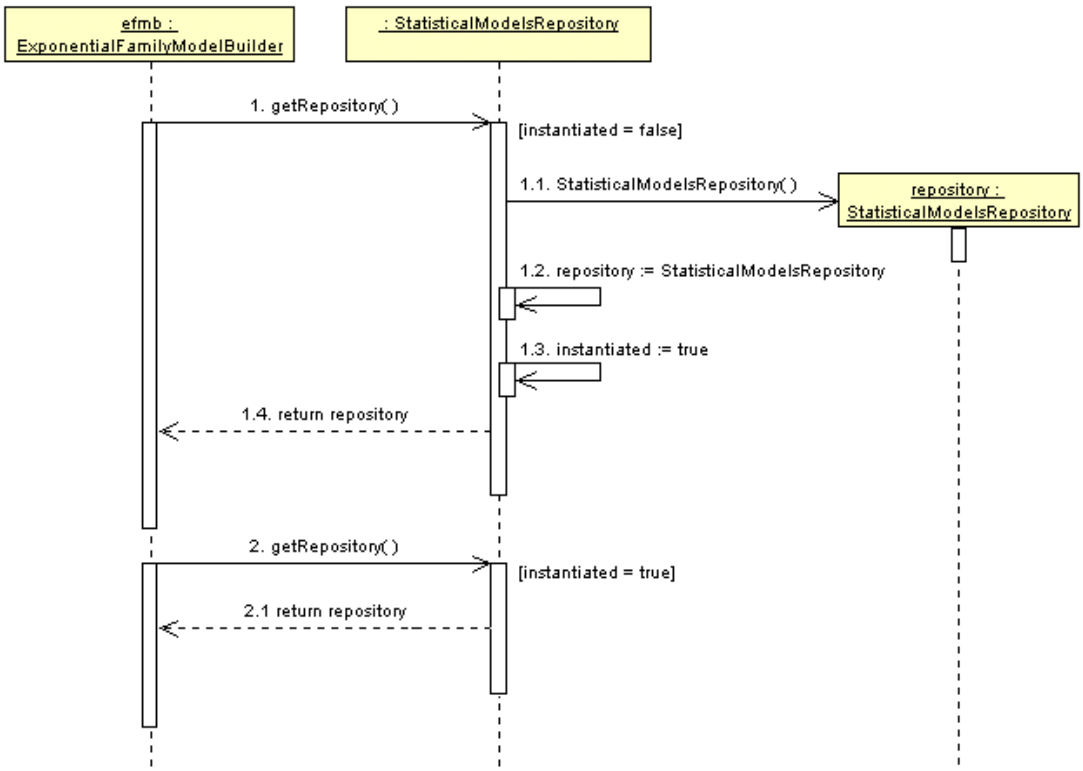


Figura 43. Diagrama de Secuencia del patrón de diseño *Singleton* aplicado a la clase *StatisticalModelsRepository*

c. La relación de agregación con Formula y ExponentialFamilyDistribution: el patrón «Strategy»

Como se vio en el subsistema anterior, las clases de tipo *Formula* y las de tipo *ExponentialFamilyDistribution* son clases que responden a un planteamiento de diseño congruente con la lógica del modelado estadístico, en el sentido de que, aunque mantienen una estrecha relación, ésta no es unívoca y, además, sus responsabilidades diferenciales permiten concebirlos como estructuras distintas. En este sentido, sus métodos y atributos se encuentran declarados en clases abstractas independientes.

El problema que se plantea en este caso, es cómo delegar estas responsabilidades en un objeto distribución, o en un objeto fórmula. Un diseñador inexperto podría recurrir directamente al uso de sentencias condicionales, de forma que si se desea implementar la funcionalidad de un tipo de distribución asociada a la clase *GLMModel*, la añadiría en esta

clase, a través de un *if*. Lo mismo haría con el resto de distribuciones de interés. Como ya se ha comentado anteriormente, esta forma de resolver el problema es típica de la programación estructurada. El inconveniente que presenta este diseño es que realmente la clase no delega responsabilidades, sino que asume dicha responsabilidad, puesto que la funcionalidad de dichas distribuciones ha sido integrada dentro de la clase. Esta solución es la antítesis de lo que se podría considerar un diseño OO, puesto que crearía clases con exceso de funcionalidad (cuando puede ser evitable), reduciría la flexibilidad del sistema (no se podría hacer uso del polimorfismo) y limitaría su extensibilidad.

Otra manera de afrontar el problema, bajo una perspectiva de orientación a objetos, consistiría en hacer uso de la herencia para conseguir, por ejemplo, que una única clase *GLMModel* adquiera la funcionalidad de diferentes distribuciones concretas, o bien de distintos tipos específicos de fórmulas. En este sentido, el diseñador, con la intención de implementar pensando en objetos derivaría de la clase *GLMModel* una nueva clase a la que podría añadir la funcionalidad de la distribución Normal, y que denominaría *GLMModelGaussian*. Si resulta que también necesita la funcionalidad de la distribución de Poisson, seguiría los mismos pasos para crear la clase *GLMModelPoisson*. De esta manera, crearía una nueva clase para cada funcionalidad concreta que desee acoplar a una clase ya existente.

Precisamente, esta última solución es la menos óptima de los posibles diseños que hacen uso de la OO, puesto que provoca un crecimiento exponencial de clases, tal como se expuso en el apartado 8.1.2. La verdadera raíz del problema de este planteamiento es que mezcla abstracción (entendida como concepto de clase) y funcionalidad, lo que hace que este sistema sea más difícil de comprender, modificar y extender.

Para evitar esta situación, se debe recurrir en este caso al patrón de diseño *Strategy* (Estrategia) (Stelting y Maassen, 2001; Gamma et al., 2003; Eckel, 2003). Este patrón define una familia de algoritmos o estrategias asociadas a una interfaz y las hace intercambiables, precisamente separando la abstracción de las diferentes funcionalidades de una clase (figura 44).

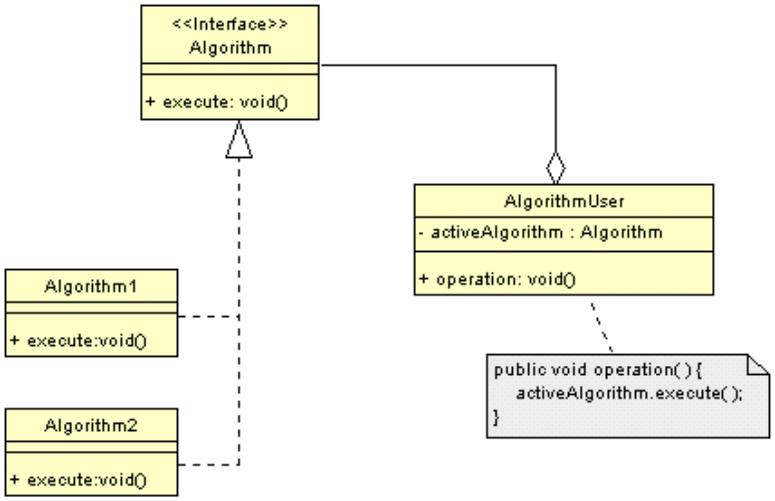


Figura 44. Patrón de diseño *Strategy*

Siguiendo con el ejemplo de diseño presentado anteriormente, basado exclusivamente en la herencia, se mezclaba en una única clase (*GLMModelPoisson*, por ejemplo) la implementación de una distribución de la familia exponencial (clase de tipo *Poisson*) con el modelo estadístico que hace uso de esta funcionalidad (clase *GLMModel*, por ejemplo). Aplicando el patrón *Strategy* se desligan totalmente ambas clases.

En efecto, como se observa en la figura 44, las clases que utilizan estas estrategias (en nuestro caso, la clase *GLMModel* o *GAMModel*), deben tener una referencia al objeto estrategia (*activeAlgorithm*) que se encuentre activo (objeto de tipo *Binomial*, por ejemplo). En este sentido, la manera de conseguir que se pueda activar cualquiera de las clases estrategia en tiempo de ejecución, es haciendo uso de una interfaz que declare el comportamiento común de éstas. Gracias a ello, se puede recurrir al uso del polimorfismo, a través de una referencia a un objeto representada genéricamente (polimórficamente) mediante el nombre de la interfaz que comparten dichas clases (atributo *activeAlgorithm* en la figura 44).

En la figura 37 se observa que las clases *ExponentialFamilyModel* tienen un atributo que es una referencia al objeto formula (atributo *formula*) y otro que referencia al objeto distribución (atributo *distribution*), y en ambos casos se ha indicado sobre las líneas de asociación el estereotipo *«Strategy»* para destacar el uso de este patrón en este caso. Estas referencias genéricas son las que precisamente permiten que las clases concretas que

implementan la interfaz –o que derivan de una clase base– sean capaces de responder a un mismo mensaje, puesto que ha sido declarado en la interfaz común o clase base referenciada. Así, el objeto *glmNegBin* de tipo *GLMModel* creado en el ejemplo anterior, haría uso internamente en su método *fit( )* del polimorfismo respecto a cualquier objeto de tipo *Formula* o de tipo *ExponentialFamilyDistribution* referenciados en los atributos señalados:

```
// Ejemplo de ejecución del método designMatrix( ) de un objeto Formula desde el método fit( ) de un modelo
ArrayList designMatrix = this.formula.designMatrix( );

// Ejemplo de ejecución del método addObserver( ) de un objeto ExponentialFamilyDistribution desde un modelo
this.distribution.addObserver( glmNegBin );
```

Estos ejemplos ilustran sin lugar a dudas la potencia del polimorfismo en los sistemas orientados a objetos diseñados haciendo uso de patrones como el *Strategy* aplicado en este subsistema. Además, cabe señalar la flexibilidad de este tipo de diseños, en el sentido que posibilitan el "enlace dinámico" entre objetos, al permitir la asignación a un objeto *ExponentialFamilyModel* de un nuevo objeto de tipo *Formula* o de un nuevo objeto de tipo *ExponentialFamilyDistribution* en tiempo de ejecución, utilizando para ello su método *update( )* –descrito al inicio de este apartado–.

8.2.2. El subsistema Ajustar Modelo

La clase *ExponentialFamilyModel*, expuesta en el anterior apartado, permite delegar a través de su método *fit( )* la responsabilidad de ajustar cualquiera de sus objetos modelo. Concretamente, delega toda esa responsabilidad en la clase *ModelFitter*, la cual se encargará de dirigir el proceso de ajuste y devolver al objeto modelo el resultado del ajuste; esto es, provoca un cambio de estado en el objeto modelo activo: pasa de ser un modelo estructural, sin ajustar, a ser un modelo ajustado.

En este subapartado nos detenemos en las clases principales del subsistema *Ajustar Modelo*: la clase *ModelFitter* y las clases que implementan la interfaz *FitterAlgorithm*. Se

justifica la presencia de estas clases, junto a otras relacionadas, así como los diferentes patrones involucrados en el diseño.

a. La clase ModelFitter y la interfaz FitterAlgorithm

Un primer problema que aparece en este contexto es cómo incluir en el sistema el algoritmo de estimación (IWLS, OLS, etc.) como elemento central en el proceso de ajuste de un modelo. En este sentido, es el mismo tipo de necesidad planteada para las clases *Formula* y *ExponentialFamilyDistribution*. El uso de la herencia para conseguir que una clase *GLMModel* integre en una clase derivada la funcionalidad de un algoritmo concreto no es la mejor opción, como ha quedado evidenciado en el apartado anterior por los motivos expuestos (explosión de clases por herencia y mezcla de abstracción y comportamiento). Puesto que se dan los mismos condicionantes, la mejor solución debería ser un diseño basado en el patrón *Strategy*.

Sin embargo, aparece un nuevo condicionante que invalida el uso de este patrón. Debido a que el preproceso y el proceso posterior a los resultados obtenidos por el algoritmo de estimación pueden variar en función del tipo de modelo que se desea ajustar (GLM, GAM, etc.), es necesario encontrar una solución que permita integrar de manera óptima esta funcionalidad. El preproceso, permite depurar y preparar los datos para que sean utilizados de forma óptima por el algoritmo concreto de estimación. A su vez, el resultado del proceso debe ser transformado para poder almacenarlo en un formato adecuado. Por tanto, debe existir un elemento intermedio entre un objeto modelo y un objeto algoritmo de estimación que asuma estas funciones. Además, la manera de implementar esta funcionalidad depende, como se ha dicho, del tipo de modelo que se ajuste. En consecuencia, debe existir un mecanismo que permita decidir, sin recurrir al uso de sentencias condicionales, el tipo de preparación de los datos.

Volvemos a encontrarnos con la necesidad de acudir al patrón de diseño *Strategy*, que en este caso supone la creación de una clase abstracta denominada *ModelFitter*, de la cual derivan las clases *GLMFitter* y *GAMFitter*, cada una de ellas especializada en una estrategia concreta de preparación de los datos previa a su uso por el algoritmo de estimación (figura 45).

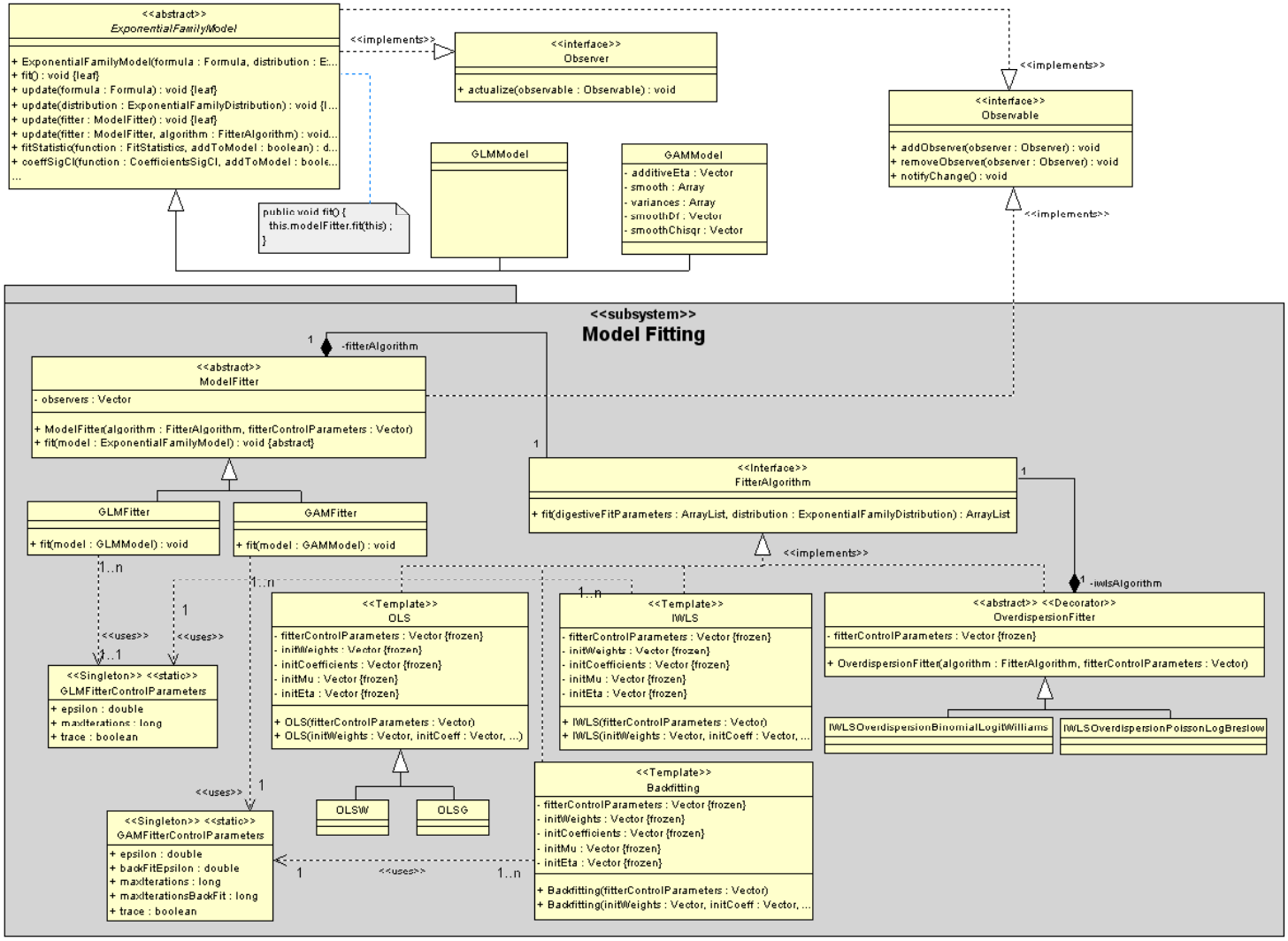


Figura 45. Subsistema *Model Fitting* y la clase *ExponentialFamilyModel*

El uso del patrón *Strategy* requiere incluir una referencia a dichas estrategias en la clase *ExponentialFamilyModel* (almacenada en el atributo *modelFitter*). En este sentido, se sigue la misma lógica que en el caso de las clases referenciadas con los atributos *formula* y *distribution* (figura 37).

Ahora bien, todavía queda un problema no resuelto. Se trata de decidir cómo se deben relacionar las clases de tipo *ModelFitter* y los algoritmos de estimación. Por descontado, queda claro que el uso de la herencia para aumentar la funcionalidad de estas clases no es la solución más adecuada, si tenemos en cuenta que existen diferentes algoritmos de estimación que además son intercambiables. En este sentido, se adopta la decisión de separar dicha funcionalidad de las clases *ModelFitter*, creando una clase diferenciada para cada algoritmo de estimación (*IWLS*, *OLS*, etc). Por otro lado, para permitir el uso del polimorfismo en tiempo de ejecución se asocia a cada una de estas clases algoritmo una misma interfaz, que denominamos *FitterAlgorithm*, y que obliga a definir un comportamiento común a cualquier algoritmo que se desee incluir en el sistema, aunque no pertenezca a una misma jerarquía de herencia que el resto de algoritmos.

El papel de control que ejerce un objeto de tipo *ModelFitter* sobre el algoritmo de estimación, desde el momento en que le indica que arranque el proceso de estimación llamando a su método *fit( )*, hasta la especificación del número de iteraciones que debe cubrir, nos hace pensar en una estrecha vinculación entre ambos, que en términos de diseño supone establecer una relación de composición entre ellos. Esto implica construir el objeto algoritmo cuando se crea el objeto de tipo *ModelFitter*, y de ello se ha encargado un objeto de la clase *ExponentialFamilyModelBuilder*, proporcionando a su vez en el constructor del objeto *ModelFitter* una referencia al objeto algoritmo. Esta relación liga el tiempo de vida de un objeto a otro.

Por otro lado, como ya se ha comentado, la funcionalidad que aporta un objeto de tipo *GLMFitter* a un objeto algoritmo consiste básicamente en preparar los datos que va a manejar el algoritmo y los datos que devuelve al final del proceso. En este sentido, parece que la clase *ModelFitter* «decora» a las clases *FitterAlgorithm*, y que puede ser adecuado utilizar en este caso el patrón de diseño *Decorator* (descrito en el apartado 8.1.2). Se dan una serie de condicionantes que hacen pensar en que efectivamente debemos aplicar este

patrón. Por un lado, el objeto *ModelFitter* modifica el producto de un objeto *FitterAlgorithm*, como de hecho podía hacer la clase *RobustExponentialFamilyDistribution* para robustecer a un objeto distribución (apartado 8.1.2), y por otro, la estructura de la relación de composición entre estos objetos es equivalente a la descrita en la figura 32 para el patrón *Decorator*.

Pero nótese que un objeto de tipo *ModelFitter* siempre modifica (procesa) el producto de un objeto algoritmo, mientras que un objeto de tipo *RobustExponentialFamilyDistribution* puede modificar (robustecer) o no a un objeto distribución. Esta diferencia de matiz es uno de los elementos que permiten discernir si es adecuado aplicar un determinado patrón u otro; en este caso es un motivo importante para justificar la inadecuación de la solución *Decorator*.

Existe un motivo adicional que impide aplicar el patrón *Decorador*: *ModelFitter* no comparte la misma interfaz (*FitterAlgorithm*) que las clases que implementan los algoritmos de estimación, aspecto éste que es un requisito de diseño necesario del patrón *Decorator*.

Por tanto, se ha de buscar una solución alternativa. Si pensamos en los algoritmos de estimación IWLS, OLS, OLSW, OLSG o Backfitting como estrategias intercambiables, quizás sea más fácil acercarnos al patrón que se debe aplicar. Además, hay que tener en cuenta que el intercambio del algoritmo en este diseño supone mucho más que una simple modificación del exterior de un objeto (propio del patrón *Decorator*); más bien supone el cambio de las «tripas» de dicho objeto. Estos elementos nos hacen pensar en la aplicación del patrón de diseño *Strategy* ya utilizado en el caso de la asociación entre la clase *ExponentialFamilyModel* y las clases *Formula*, *ExponentialFamilyDistribution* y *ModelFitter*. De hecho, ésta ha sido la solución de diseño finalmente adoptada, decisión altamente recomendable, por otra parte, si el algoritmo de estimación se encuentra almacenado en librerías estadísticas y matemáticas estandarizadas, como *NAG*, *IMSL* o *Numerical Recipes* (Press, Flannery, Teukolsky y Vetterling, 1989).

En consecuencia, queda justificado en la imagen del subsistema (figura 45), que la clase *ModelFitter* debe asumir la responsabilidad del control del proceso de ajuste, y que esta delegación de responsabilidad en un tipo concreto de estrategia (*GLMFitter* o *GAMFitter*)

se realiza a partir del patrón *Strategy*. Por otro lado, también se ha justificado que los diferentes algoritmos de estimación que permiten procesar el ajuste deben ser escogidos a través de la clase *ModelFitter*, volviéndose a hacer uso del patrón *Estrategy* para cubrir esa necesidad –*véase* anexo para una imagen completa del DCD del subsistema *Model Fitting*.

Cuando se ejecuta el método *fit( )* de un objeto de tipo *ExponentialFamilyModel* para iniciar el proceso de ajuste del modelo, éste invoca a su vez la ejecución del método *fit( )* de un objeto de tipo *ModelFitter* enviando a través de su signatura una referencia a sí mismo –*véase* la nota de implementación en figura 45, en la que se observa que el método *fit( )* del objeto modelo se pasa a sí mismo (*this*) como argumento del método *fit( )* del objeto *ModelFitter*–. De esta manera, el objeto *ModelFitter* tendrá acceso a todos los métodos y atributos públicos del objeto *ExponentialFamilyModel*, y a los métodos y atributos públicos de los objetos referenciados en este último (de tipo *Formula* y *ExponentialFamilyDistribution*). De este modo, el objeto *ModelFitter* consigue toda la información necesaria para realizar su tarea de preproceso previa a la invocación del objeto algoritmo de estimación (necesita utilizar, entre otros, los métodos del objeto *Formula* asociado al modelo para extraer los vectores y matrices de datos pertinentes). En Java, la sentencia que realiza esta labor tendría la forma:

```
// Instanciación del objeto modelo estadístico (GLMModel) con agregación de los objetos Formula,
// ExponentialFamilyDistribution y ModelFitter (que incluye un objeto de tipo FitterAlgorithm)
GLMModel glmBinom =
        new GLMModel( new Formula( “y ~ x1 * x2 “ ), new BinomialLogit(), new GLMFitter( new IWLS( null ) );
// Invocación del proceso de ajuste del modelo
glmBinom.fit( );
. . .
// Llamada interna desde el método fit( ) del objeto glmBinom al método fit( ) del objeto GLMFitter
// referenciado en el atributo modelFitter de glmBinom
        . . .
        modelFitter.fit( this );
        . . .
// Uso interno desde el método fit( ) del objeto ModelFitter de los métodos y atributos del objeto Formula
// referenciado en el atributo formula de glmBinom
        . . .
```

```
ArrayList data = model.getFormula().dataModel();
Vector y = (Vector) model.getFormula().extractTerm( data, Formula.FORMULA_RESPONSE );
ArrayList dm = model.getFormula().designMatrix();
. . .
// Llamada al algoritmo de estimación IWLS desde el método fit( ) del objeto GLMFitter
. . .
fitterAlgorithm.fit( new ArrayList( y, dm, ...), model.getDistribution() );
```

Como se ha detallado anteriormente, y tal como se puede observar en el ejemplo de implementación anterior, cuando se instancia un nuevo objeto de tipo *ModelFitter*, su constructor recibe como parámetro un objeto algoritmo concreto (IWLS en el ejemplo). De esta manera se vincula fuertemente a este último, para delegar en él la responsabilidad correspondiente. En ese sentido, el método *fit( )* de un objeto *FitterAlgorithm*, llamado desde el objeto de tipo *ModelFitter*, recoge como parámetro un objeto de la familia de distribuciones, puesto que necesitará enviar mensajes a este último para delegarle, entre otras, las responsabilidades de cálculo relacionadas con la función de enlace, la función de variancia, o la obtención de los residuales de discrepancia.

Otro aspecto a considerar, en relación a la funcionalidad de las clases de tipo *ModelFitter*, es el hecho de que necesitan acceder a información sobre el número máximo de iteraciones del proceso, la traza del proceso (información resumen de los resultados en cada iteración), y determinar si desea controlar el valor de *epsilon* (criterio de convergencia). Los valores de estos atributos son distintos para cada tipo de modelo (GLM o GAM), por lo que deben ser almacenados en puntos distintos. La clase *IWLS* y la clase *Backfitting* –algoritmo de estimación descrito en Chambers y Hastie (1991, pp. 210-213)– también deben tener acceso de manera diferenciada a estos criterios. En este sentido, el algoritmo *Backfitting* necesita de unos criterios de control adicionales (*backFitEpsilon* y *maxIterationsBackFit*).

Una solución adecuada a esta necesidad consiste en acceder a variables globales. En este sentido, los atributos estáticos representan un punto de acceso global. Este requisito puede ser resuelto de manera óptima con el patrón de diseño *Singleton* (descrito en el apartado 8.2.1.b). Con esa intención se introducen en el sistema dos clases *Singleton* (figura 45): la clase *GLMFitterControlParameters*, para definir los atributos que recogen esta

información de control a los que accederán los objetos de tipo *GLMFitter* y los de tipo *IWLS*, y la clase *GAMFitterControlParameters*, que contiene los atributos a los que accederán los objetos de tipo *GAMFitter* y de tipo *Backfitting*.

La intención de estas dos clases es la de proporcionar unos valores criterio por defecto, accesibles de manera global (estática). Ahora bien, como se puede observar en la interfaz definida en las clases *ModelFitter* y en las clases algoritmo, éstas recogen la posibilidad de recibir otros valores través de sus constructores (parámetro *fitterControlParameters*).

Un aspecto funcional de diseño ya comentado anteriormente, y que ofrece la flexibilidad necesaria de cara al uso del *framework* en el contexto de la investigación empírica mediante simulación estocástica, ha sido la inclusión del patrón *Observer* (*véase* apartados 8.1.1 y 8.2.1). En este sentido, las clases *GLMFitter* y *GAMFitter* de este subsistema implementan, al igual que lo hacen las clases *Formula* y *ExponentialFamilyDistribution* del subsistema *Especificación del Modelo*, la interfaz *Observable* (figura 45), con la finalidad de almacenar referencias a los objetos observadores e implementar métodos para enviar solicitudes de actualización a los mismos cuando cambie su estado.

También se ha considerado la inclusión en este subsistema de una funcionalidad importante para el modelado de la sobredispersión. Concretamente, se añaden las clases *IWLSOverdispersionBinomialLogitWilliams* y *IWLSOverdispersionPoissonLogBreslow*, ambas derivadas de la clase *OverdispersionFitter*. Los constructores de estas clases reciben como parámetro un objeto de tipo *FitterAlgorithm*. La intención es poder intervenir en el proceso de ajuste implementado por un algoritmo activo para un modelo dado, para adaptar dicho ajuste a aquellos datos en los que ha sido detectada la presencia de sobredispersión. Existe una fuerte vinculación entre un algoritmo y un objeto de tipo *OverdispersionFitter*, puesto que este último basa su trabajo en los resultados arrojados por el primero. En este sentido, esta relación es similar a la que existe entre un objeto de tipo *ModelFitter* y un objeto *FitterAlgorithm*, puesto que el primero se encarga de procesar la información anterior y posterior a la ejecución del segundo. Además, la estructura de ambos tipos de relaciones es muy parecida.

Si nos basamos únicamente en estas características, parece ser que deberíamos volver a aplicar el patrón de diseño *Strategy*. Ello conllevaría el mismo error que se planteaba en un

caso anterior. Siempre existen diferencias clave entre patrones de diseño similares. De hecho, se puede recordar la discusión que se planteaba en este sentido para decidir el patrón de diseño a aplicar entre las clases de tipo *ModelFitter* y las clases de tipo *FitterAlgorithm*. Estamos ante el mismo problema, pero esta vez nos decidimos por el patrón *Decorator* (figura 45). Los motivos son claros, y van en la siguiente dirección: las clases de tipo *OverdispersionFitter* simplemente «decoran» el producto del ajuste de un modelo para corregir la presencia de sobredispersión; dicho de otro modo, envuelven al objeto algoritmo, sin modificar su núcleo. En cambio, en la relación entre un algoritmo y un objeto *ModelFitter*, este último modifica su núcleo a partir de diferentes estrategias de estimación. Esta diferencia es fundamental. En este sentido, un objeto algoritmo ha de actuar necesariamente en el proceso de ajuste, mientras que la intervención de un objeto que modele la sobredispersión sólo tendrá sentido precisamente cuando ésta sea detectada. Por último, nótese que las clases algoritmo y las clases de tipo *OverdirpersionFitter* comparten una misma interfaz, la recogida en la clase *FitterAlgorithm*, característica ésta que define la estructura del patrón *Decorator*.

b. La interfaz FitterAlgorithm y el patrón «Plantilla»

En este subapartado, planteamos una discusión sobre un aspecto que ya ha sido descrito anteriormente. Nos referimos a la función que ejerce un algoritmo en el ajuste del modelo, el cual necesita nutrirse de la funcionalidad proporcionada por las clases de la familia exponencial de distribuciones. Como se comentó, su método *fit( )* recoge como parámetro un objeto distribución. Esto permite al algoritmo solicitar de manera directa al objeto distribución que le proporcione la funcionalidad necesaria durante el proceso de ajuste. Por tanto, el algoritmo presenta una estructura de comportamiento global fijo, con una serie de pasos u operaciones bien establecidos, pero la funcionalidad de dichos pasos difiere en función de la distribución. Esta configuración puede recordar al patrón *Strategy*, puesto que existen una serie de elementos que varían en el objeto que centraliza la ejecución de un determinado proceso.

Ahora bien, en este caso no cambia una estrategia o algoritmo concreto, sino que cambia la manera de resolver determinados pasos de una estrategia concreta. En definitiva, se redefinen ciertos pasos de un algoritmo sin cambiar su lógica de funcionamiento (su

estructura interna), mientras que en el patrón *Strategy* lo que cambia precisamente es dicha estructura. En este sentido, existe un patrón de diseño ampliamente utilizado, que cubre precisamente la resolución de un problema en el que aparece una estructura fija que ha sido configurada para concretar la resolución de determinados pasos que pueden variar, y que necesariamente han de ser implementados. Es el patrón *Template* (Plantilla), conocido también como *Template Method* (Stelting y Maassen, 2001; Gamma et al., 2003; Eckel, 2003). Wirfs-Brock et al. (1990), proporcionan una buena discusión sobre este patrón de diseño.

El diseño del patrón *Template* (figura 46) se caracteriza por definir el esqueleto de un algoritmo en una operación denominada genéricamente "*método plantilla*" (normalmente implementada en una clase abstracta), delegando en subclases algunas de sus operaciones; cada subclase redefine de una manera concreta dichas operaciones.

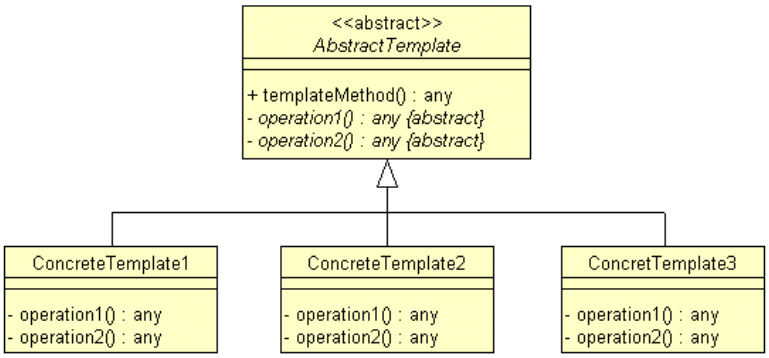


Figura 46. El patrón de diseño *Template*

El diseño abstracto de este patrón representa una manera general de solucionar el problema planteado –como ocurre en la mayoría de patrones–. En este sentido, se pueden valorar posibles alternativas de diseño concretas, siempre y cuando se mantenga la filosofía del patrón, es decir, la forma general de resolver el problema al que responde.

En nuestro caso, cada uno de los algoritmos de estimación representa un patrón *Template*, puesto que define una estructura fija, el núcleo del algoritmo, y una serie de operaciones que pueden variar (aplicación de la función de enlace, de la función de variancia, etc.). Si

nos basáramos en el diseño abstracto mostrado en la figura 46 se deberían derivar de cada una de las clases de tipo *FitterAlgorithm* tantas subclases como implementaciones distintas de cada una de las operaciones que solicita el algoritmo y que están asociadas a un tipo concreto de familia de distribución exponencial. Por supuesto, incurriríamos en el problema conocido de la explosión de clases. Además, perdería su sentido el diseño planteado en el subsistema *Especificación del Modelo*, puesto que gran parte del mismo recoge precisamente las operaciones que necesita el algoritmo de estimación para ajustar un objeto modelo.

Por tanto, hay que encontrar una solución alternativa al diseño genérico del patrón *Template* planteado en la figura 46. En realidad, esta solución ya ha sido explicada en el desarrollo de este apartado, aunque no de manera explícita. Consiste en proporcionar al objeto algoritmo una referencia al objeto distribución a través de los parámetros de su método de ejecución *fit( )* (*véase* interfaz *FitterModel*, figura 45). De esta manera, el objeto *FitterAlgorithm* podrá llamar a las funciones asociadas al objeto distribución, para que éste le devuelva la información solicitada. De aquí se desprende que el uso que se ha dado al patrón *Template* en esta parte del diseño optimiza la solución, puesto que aprovecha una estructura ya establecida, que además permite un uso exquisito del polimorfismo.

En la figura 47 se muestra el diagrama de secuencia del patrón de diseño *Template*, que ofrece una vista dinámica de la relación entre los objetos involucrados en este patrón.

En la figura 48 se muestra además, tras la descripción y justificación de cada uno de los elementos del subsistema *Ajuste del Modelo*, una vista dinámica del flujo global de mensajes que se establece entre los objetos involucrados en dicho subsistema.

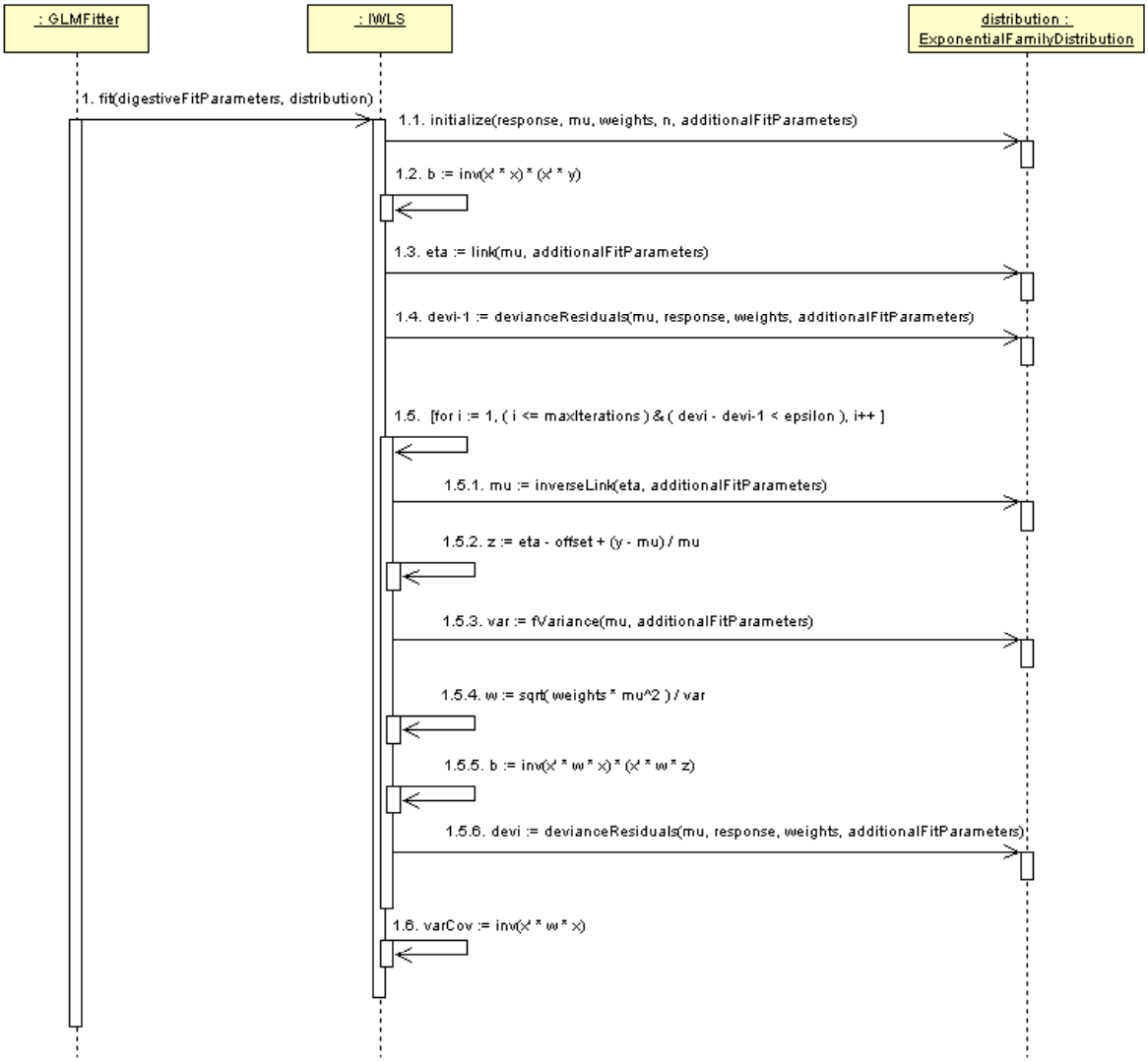


Figura 47. Diagrama de Secuencia del patrón *Template*

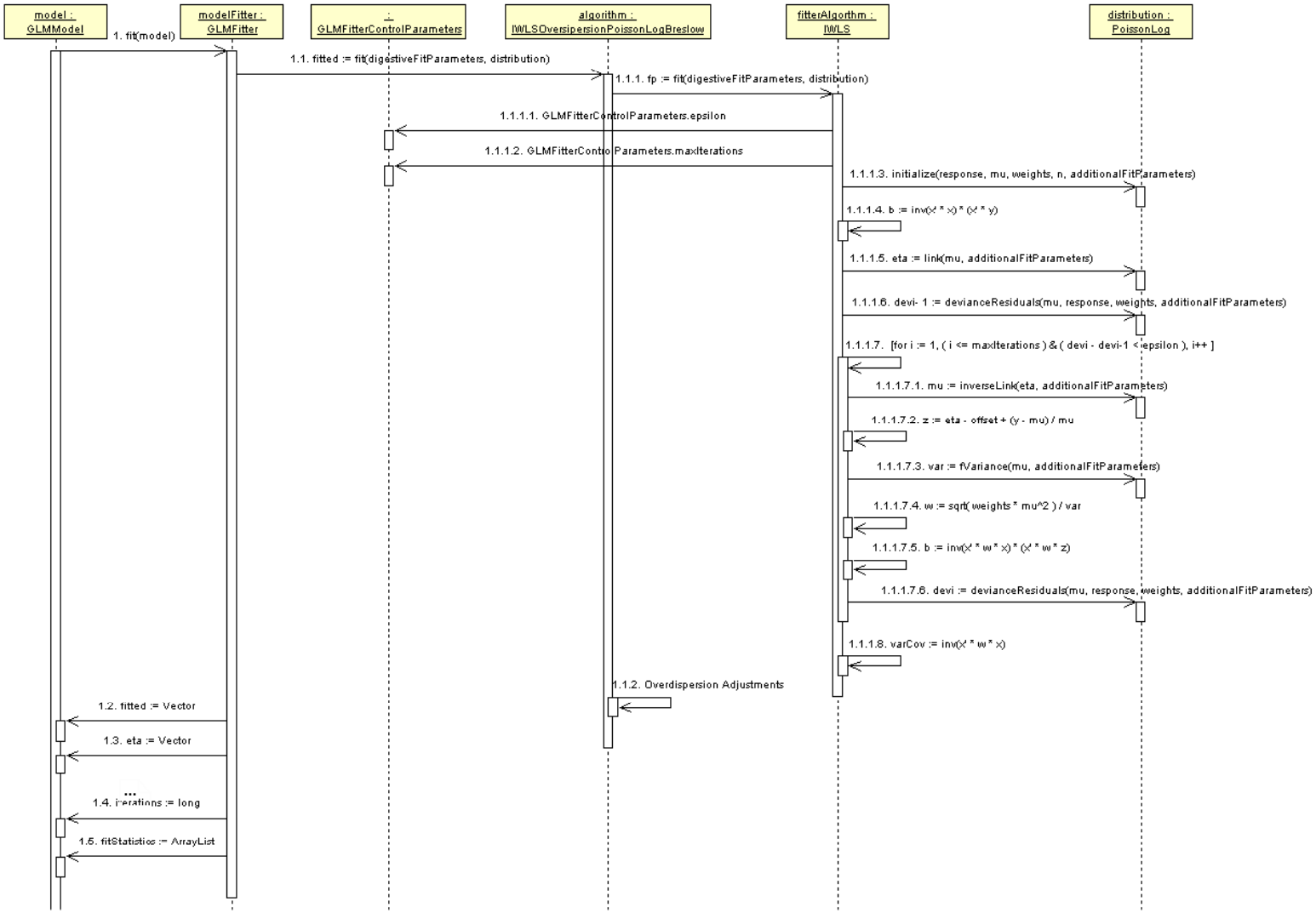


Figura 48. Diagrama de Secuencia del subsistema *Model Fitting*

c. Las clases FitStatistics y CoefficientsSigCI y el patrón «Visitor»

Tal como se vio en la introducción del apartado 8.2.1, los intervalos de confianza y la significación estadística de los coeficientes de un modelo ajustado, así como el acceso a otros estadísticos de ajuste, se recogen en las clases de tipo *ExponentialFamilyModel*.

Un problema que se plantea en este sentido es cómo integrar en el diseño del sistema el cálculo de estos elementos, teniendo en cuenta que su funcionalidad es utilizada por las clases de tipo *ExponentialFamilyModel*, y que no es viable implementarlas haciendo uso de la herencia. Se cuenta con diversos estadísticos para solventar estas demandas y no todos ellos se calculan de la misma manera, aspecto que depende del tipo de modelo ajustado (GLM o GAM). Por otra parte, estos estadísticos se pueden agrupar en función de su papel como elementos criterio para valorar la significación estadística y práctica (*t* de Student, *F* de Snedecor, etc. e intervalos de confianza) de los coeficientes del modelo, y por otro lado, aquellos estadísticos que cubren el análisis de bondad de ajuste del modelo (Discrepancia, R^2, AIC, BIC, etc.).

El uso del patrón de diseño *Strategy* no está justificado en este caso, puesto que dichos elementos estadísticos no son objetos que modifiquen el interior del objeto modelo (su núcleo pesado de resultados de ajuste). De ello se ha encargado un objeto de tipo *ModelFitter*. En este caso, el modelo accede a determinadas funciones estadísticas para que le proporcionen cierta funcionalidad extra, que además es opcional. En este sentido, sabemos que el patrón *Decorador* permite ampliar la funcionalidad de un objeto decorable siempre y cuando comparta la misma interfaz que el objeto que lo decora. Dado que estos elementos no comparten dicha interfaz, ya que no tiene sentido que los estadísticos de bondad de ajuste y de significación de los coeficientes del modelo sean apilables (característica propia de los decoradores), descartamos su uso en el diseño.

Otro de los patrones conocidos, que permite añadir nueva funcionalidad sin cambiar las clases de los elementos sobre los que opera es el patrón *Visitor* (Visitante), descrito en Stelting y Maassen (2001), Gamma et al. (2003) y Eckel (2003). Este patrón utiliza el concepto de objeto *visitante*, que aporta funcionalidad extra a un objeto *visitado* (figura

49). Los requisitos que debe asumir el sistema para diseñar la solución a este problema, indican que éste es el patrón de diseño adecuado.

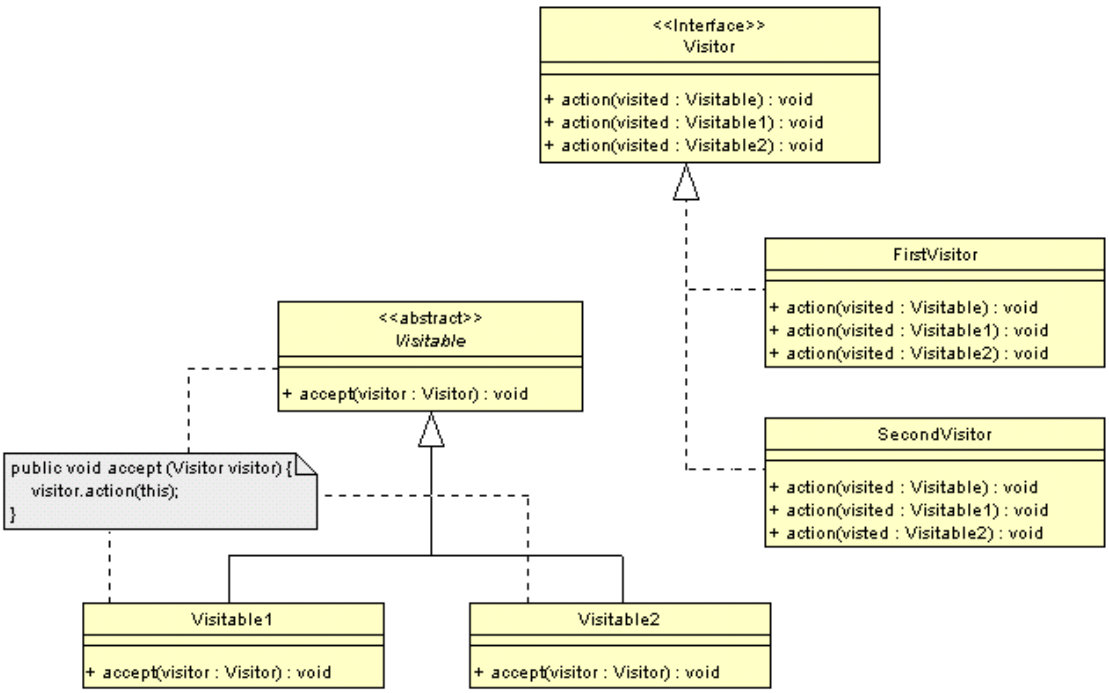


Figura 49. Estructura general del patrón de diseño *Visitor*
(Fuente: Ocaña y Sánchez, 2003)

Como se observa en la figura anterior, las clases de tipo *Visitable*, tienen un método *accept( )* que recibe como parámetro un objeto de la clase de tipo *Visitor*. Por otro lado, las clases *Visitor* implementan un mismo método (de manera distinta) para cada una de las clases visitables. En ese sentido, cada uno de estos métodos tiene que recoger como parámetro la clase visitable a la que afecta dicha implementación. El control de la ejecución de la funcionalidad implementada en una clase visitante la asume la clase visitable, a través de su método *accept( )*, cuya implementación es la misma para cada una de ellas (*véase* nota en la figura 49). La manera en que un objeto visitable le indica a un objeto visitante que debe ejecutar una implementación concreta de entre sus métodos sobrecargados, es informándole del tipo de objeto que es. En este caso, para aprovechar la capacidad polimórfica en tiempo de ejecución existe la posibilidad de utilizar el parámetro *this* en Java, o *self* en C++.

En nuestro diseño (figuras 50 y 51), se han considerado dos grupos de clases visitantes diferenciadas de manera jerárquica. Por un lado, la familia de clases derivadas de *CoefficientsSigCI,* y por el otro las derivadas de la clase *FitStatistics*. Cada una de ellas utiliza la sobrecarga de métodos –método *calculate( )*– para implementar el cálculo de la funcionalidad correspondiente, en base al tipo de modelo ajustado (GLM o GAM). A su vez, cada una de las clases visitables implementa (de la misma manera) los métodos *accept( )* de llamada a las funciones de las clases visitantes –en este caso, nos referimos a los métodos *coeffSigCI( )* y *fitStatistic( )* de la clases derivadas de *ExponentialFamilyModel*, respectivamente–. Un ejemplo de uso de estos métodos sería el siguiente:

```
// Instanciación del objeto modelo estadístico (GLMModel) con agregación de los objetos Formula,
// ExponentialFamilyDistribution y ModelFitter (que incluye un objeto de tipo FitterAlgorithm)
GLMModel glmPoisson =
        new GLMModel( new Formula( “y ~ x1 * x2 “ ), new PoissonLog(), new GLMFitter( new IWLS( null ) );
// Invocación del proceso de ajuste del modelo
glmPoisson.fit( );

// Cálculo de la significación estadística y de los IC de los parámetros del modelo
Array coef = glmPoisson.coeffSigCI( new ZWald(), true );

// Cálculo de diferentes índices estadísticos de bondad de ajuste
double pseudoR2 = glmPoisson.fitStatistic( new PseudoR2(), true );
double pearsonChiSqr = glmPoisson.fitStatistic( new X2Pearson(), true );
double cpMallows = glmPoisson.fitStatistic( new CpMallows(), true );
```

El punto más débil de este patrón es el hecho de que la interfaz *Visitor* debe incluir métodos para todas las clases que tengan que ser visitadas (en este caso, las clases *GLMModel* y *GAMModel*). Concretamente, aparecen los métodos sobrecargados *calculate(model:GLMModel*) y *calculate(model:GAMModel*) en cada una de las clases *Visitor*. En este sentido, si se amplía la jerarquía de la clase base *ExponentialFamilyModel*

con un nuevo tipo de modelo, habría que incluir en cada una de las clases visitantes un nuevo *calculate( )* que referencie a este nuevo objeto visitable. En este caso, sería necesario retocar el código de estas clases. Dentro de lo que cabe esto no es tan grave, puesto que la jerarquía de la clase *ExponentilaFamilyModel* será normalmente más estable que la jerarquía de las clases visitantes (*CoefficientsSigCI* y *FitStatistics*). De hecho, esta última estará bajo control del usuario, que la utilizará como vía de ampliación de la primera. Por ejemplo, la adición de un nuevo estadístico de bondad de ajuste (que derive de la clase *FitStatistics*) se integra de manera directa en este diseño, simplemente derivando una nueva clase de la clase base, y sin la necesidad de realizar ningún otro tipo de cambio en la implementación.

Nos detenemos en otras características concretas de nuestro diseño. Las clases base *CoefficientsSigCI* y *FitStatistics* han sido consideradas clases abstractas (y no interfaces, como en el esquema general de la figura 49), con la finalidad de poder contener atributos, que en este caso permite que sus clases derivadas almacenen el resultado de su ejecución.

Por otro lado, las clases visitables (*GLMMode*l y *GAMModel*) recogen los atributos *fitStatistics* y *coefSig* (de tipo *Vector*), con la finalidad de almacenar las referencias a cada uno de los objetos visitantes que han sido llamadas desde los métodos *fitStatistic( )* y *coeffSigIC( )*, respectivamente. De esta manera, se recoge la posibilidad de acceder desde el propio objeto modelo a un resumen de los resultados almacenados en los objetos visitantes.

Nótese que los métodos *coeffSigIC( )* y *fitStatistic( )* de las clases derivadas de *ExponentialFamilyModel*, además de contener un argumento que referencia a los visitantes que proporcionan la funcionalidad de interés, también contienen un argumento (*addToModel*), de tipo *boolean*, que permite indicar si se está interesado en registrar estos objetos visitantes como objetos observadores. En este sentido, los objetos observadores serán almacenados en el atributo *observers* de la clase visitable *GLMModel* o *GAMModel* (*véase* patrón *Observer*, apartados 8.1.1 y 8.2.1).

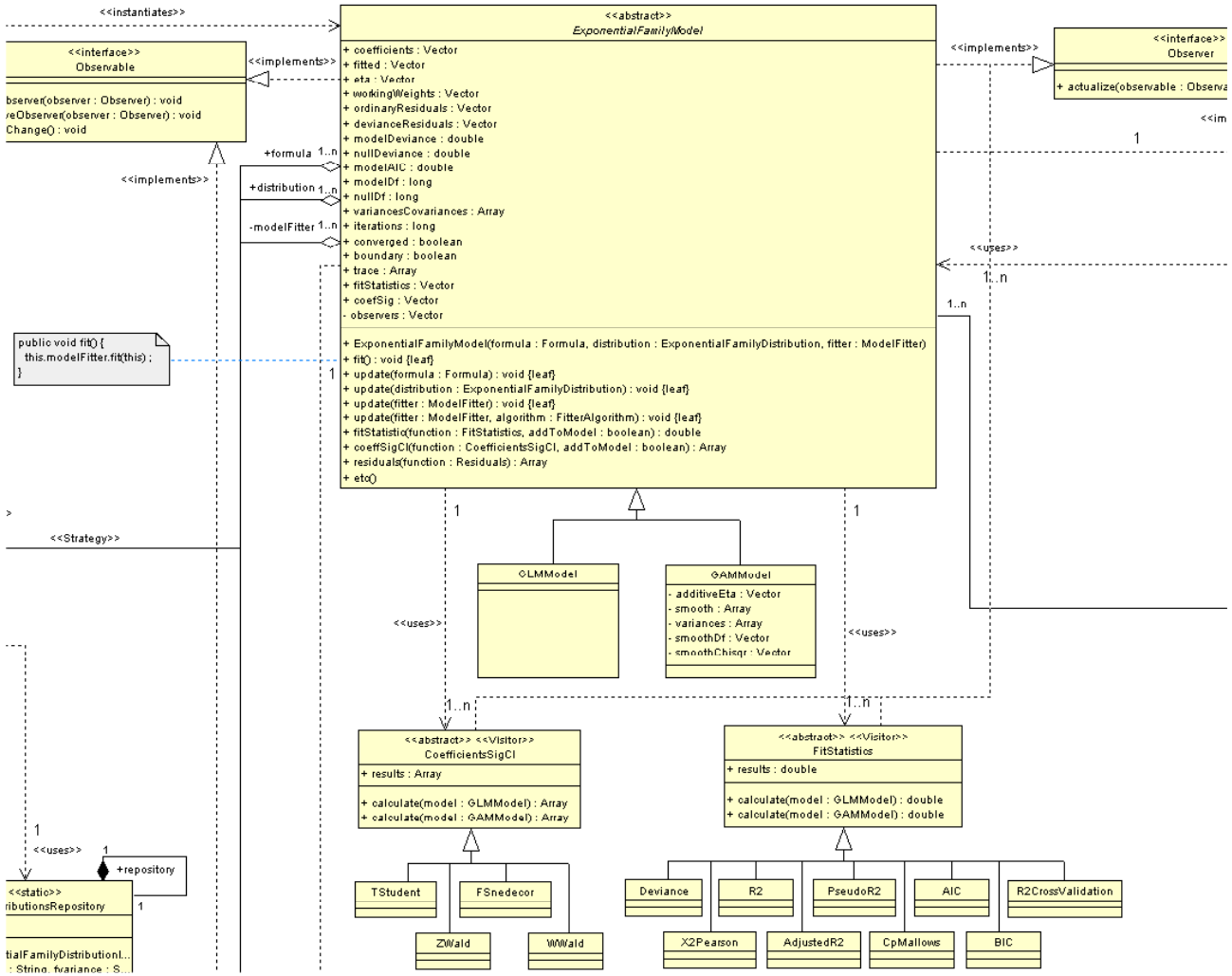


Figura 50. Diagrama de Diseño del patrón *Visitor* para la obtención de la significación y el intervalo de confianza de los coeficientes del modelo, y para el cálculo de los índices de bondad de ajuste

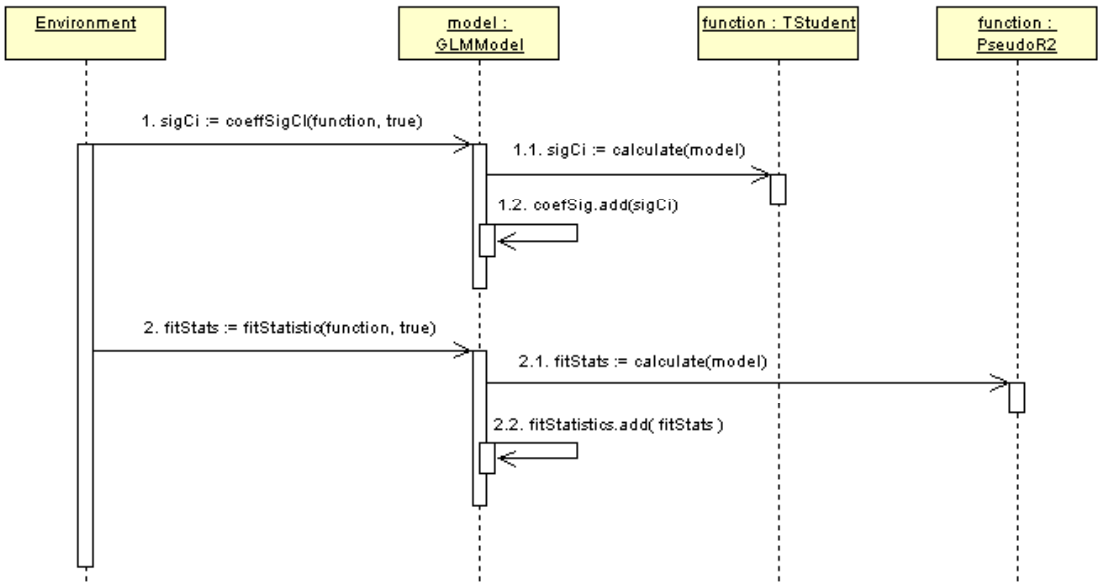


Figura 51. Diagrama de Secuencia de la aplicación del patrón *Visitor* para la obtención de la significación y el intervalo de confianza de los coeficientes del modelo, y para el cálculo del índice de bondad de ajuste pseudo-R^2

8.2.3. El subsistema Comparar Modelos: la clase ModelComparison y el patrón «Polimorfic Factory»

El subsistema *Model Comparison* necesita de un mecanismo que automatice la creación de objetos que evalúen la significación estadística de la diferencia de discrepancias de dos objetos modelo. Estos objetos deberían ser de tipo *F* o *ChiSqr*, en principio. En este sentido, se requiere flexibilidad en la elección en tiempo de ejecución del tipo de objeto evaluador que se desea instanciar, esto es, evitar tener que indicar en el código del sistema el nombre de la clase asociada a un tipo concreto de objetos, lo cual anularía la posibilidad de uso del polimorfismo.

El patrón *Polimorfic Factory* (*Factoría Polimórfica*), descrito en Eckel (2003) y en Gamma et al. (2003)[4], resuelve esta necesidad. El propósito de este patrón es encapsular la creación de objetos en otro objeto que asume el papel de *factoría*. Este patrón presenta una

[4] Estos autores denominan *Factory Method* (*Método de Fabricación*) al patrón *Polimorfic Factory*.

estructura en la que existe una clase que contiene el método de fabricación (la clase *Factory*) que devuelve un objeto concreto (figura 52).

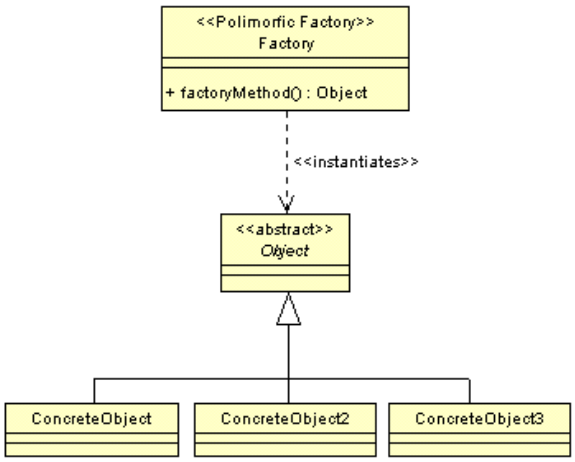


Figura 52. Patrón *Polimorfic Factory*

La clase abstracta *Object* declara el comportamiento y atributos de los objetos que ha de crear el método *factoryMethod( )* de la clase *Factory*, esto es, permite que sus clases derivadas especifiquen la implementación concreta de dicha funcionalidad. De esta manera, los métodos de fabricación eliminan la necesidad de ligar clases específicas de la aplicación a nuestro código, puesto que dicho código sólo trata con el comportamiento declarado en la clase abstracta. Dicho de otro modo, la clase *Factory* puede crear instancias de objetos específicos sin conocer sus clases, siempre que estás implementen el comportamiento declarado por la clase base *Object*. Esta estructura otorga flexibilidad al sistema, puesto que se pueden crear objetos de un tipo concreto mediante la llamada al método *factoryMethod( )* de manera polimórfica. Por tanto, este patrón de diseño cubre el requisito que se marcaba al principio de este subapartado. Esta flexibilidad no sería posible si los objetos concretos que se crean no derivan de una clase base (comparten su comportamiento).

Una visión más general de este patrón (Eckel, 2003; Gamma et al., 2003) parte de una clase *Factory* abstracta, que declara el método *factoryMethod( )* de forma abstracta, lo cual

obliga a derivar clases *Factoría* concretas que implementen dicho método para instanciar objetos de distinto tipo. En otras palabras, cada clase *Factory* concreta se especializa en la creación de un tipo concreto de objetos (que comparten un mismo comportamiento). Esta variante es útil cuando no se puede prever el tipo de objetos que se han de crear.

En nuestro caso (figura 53), no acudimos a una clase *Factoría* abstracta puesto que el tipo de objetos que se van a crear ya han sido previstos, serán objetos de tipo *ModelComparison* (clase abstracta), razón por la cual se define directamente una factoría concreta (*ModelComparisonFactory*). Por tanto, el patrón *Polimorfic Factory* interviene en este subsistema con la finalidad de encapsular el proceso de creación de los objetos de tipo *ModelComparison* (clases *F* y *ChiSqr*).

Por otra parte, también se incluye un repositorio que registra las diferentes clases derivadas de *ModelComparison,* siguiendo la misma lógica planteada para otras clases del sistema (figura 40, apartado 8.2.1.b), para evitar de este modo que la ampliación del *framework* por esta vía suponga tener que alterar el código de la clase *ModelComparisonFactory*.

El método *getComparison( )* (método de fabricación) recibe como parámetros un vector que contiene los modelos a comparar (parámetro *models*) y una cadena de caracteres que indica el tipo de prueba a emplear en la comparación (*F* o *ChiSqr*). Por tanto, gracias al conocimiento que tiene este método de fabricación del tipo de prueba a emplear, se encargará de llamar al constructor de la clase derivada de *ModelComparison* que sea pertinente, pasándole como parámetro el objeto *models* que había recibido: *F(models: Vector)* o *ChiSqr(models: Vector)*. Una vez construido el objeto de tipo *F* o *ChiSqr*, se ejecutará su método *fitDifference( )* para obtener el array que contendrá los resultados de la comparación. Por ejemplo, el código Java para instanciar un nuevo objeto del tipo *ModelComparison* y obtener la significación estadística de la diferencia entre las discrepancias de dos modelos (m1 y m2, en este caso) sería el siguiente:

```
// Instanciación de un objeto del tipo ModelComparison para realizar el test ChiSqr de diferencia
// de discrepancias entre dos modelos
ModelComparison mc = new ModelComparisonFactory( ).getComparison(  new Vector( m1, m2 ), “ChiSqr” );
Array compareM1_M2 = mc.fitDifference( );
```

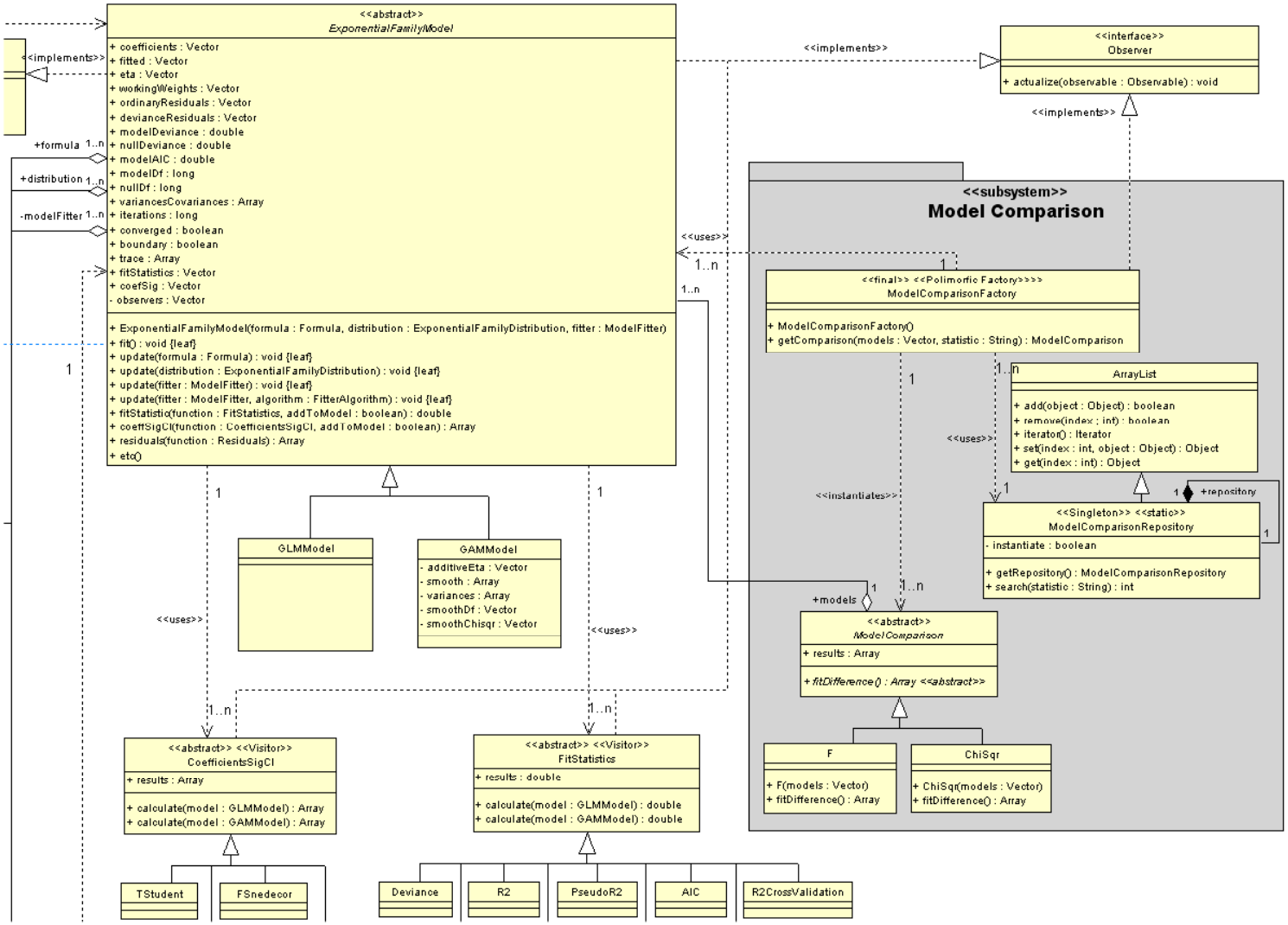


Figura 53. Diagrama de Diseño del subsistema *Model Comparison*

Siguiendo una coherencia de diseño, se introduce el patrón *Observer* en este subsistema, para conseguir que la clase *ModelComparisonFactory* sea sensible a los cambios de estado asociados a la especificación del modelo o al proceso de ajuste del mismo. Al implementar la interfaz *Observer*, el sistema asegura que el estado de un objeto *F* o *ChiSqr* cambie cuando lo hagan los objetos a los que observa el objeto *factory* –siempre y cuando éste haya sido incluido en el vector *observers*.

8.3. LOS SUBSISTEMAS EVALUAR E INTERPRETAR MODELO

Las clases de los subsistemas *Model Evaluation* y *Model Predictions* (figura 54) no aportan ninguna novedad importante a nivel de diseño. Como ocurría con las clases de tipo *CoefficientsSigCI* y *FitStatistics*, las clases de estos dos subsistemas están también ligadas a las clases de tipo *ExponentialFamilyModel* a través del patrón *Visitor*.

Sin embargo, puesto que en este caso se ha decidido que dichas clases no guarden memoria de sus resultados, éstas adquieren su comportamiento directamente de interfaces, y no de clases abstractas. Esto es, no se requiere que tengan atributos (estado), sólo ofrecen funcionalidad. Cada vez que se desee obtener información sobre algún aspecto relacionado con la evaluación del modelo o sobre predicciones concretas, bastará con llamar al método *calculate( )* ligado a la clase de interés, pasándole como parámetro un objeto modelo, tal como muestra el siguiente ejemplo de uso:

```
// Instanciación del objeto modelo estadístico GLMModel e invocación del proceso de ajuste del modelo
GLMModel glmPoisson =
        new GLMModel( new Formula( “y ~ x1 * x2 “ ), new PoissonLog( ), new GLMFitter( new IWLS( null ) );
glmPoisson.fit( );
// Obtención de diferentes índices de evaluación del modelo y predicciones
Array jackknifeResid = new JackknifeResiduals( ).calculate( glmPoisson );
Array levers = new Levers( ).calculate( glmPoisson );
Array dCook = new DCook( ).calculate( glmPoisson );
Array boot = new BoostrapEstimation( ).calculate( glmPoisson );
Array pred = new PredictionIntervalFitted( ).calculate( glmPoisson );
```

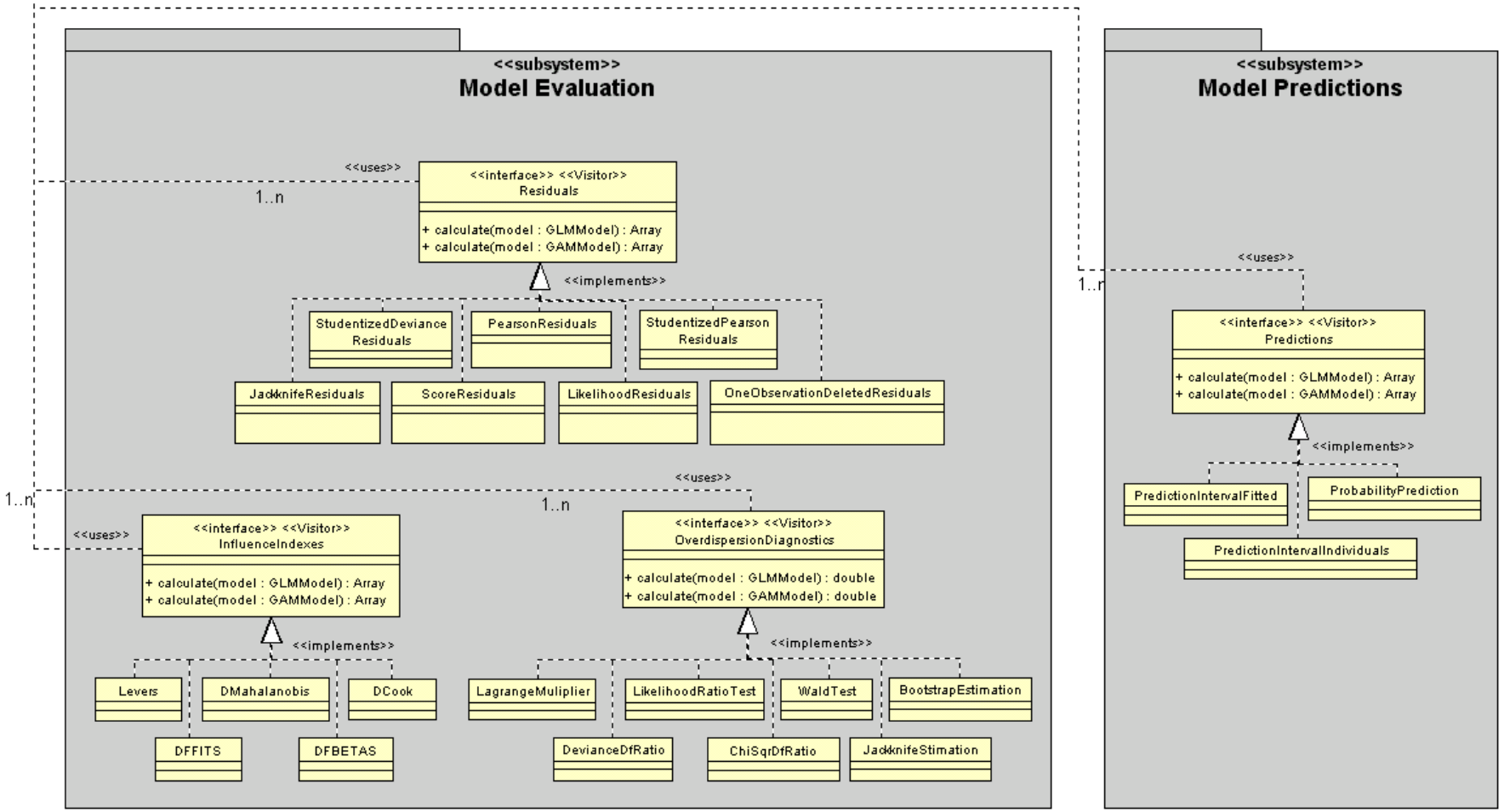


Figura 54. Diagrama de Diseño de los subsistemas *Model Evaluation* y *Model Predictions* (las líneas punteadas señalan la relación de uso de las clases de tipo *ExponentialFamilyModel* desde las clases de estos subsistemas)

9. DISCUSIÓN FINAL Y CONCLUSIONES

En este trabajo se ha intentado mostrar el panorama actual en el que se encuentra el desarrollo de software en el campo de la ingeniería informática, una perspectiva de desarrollo basada en el paradigma de la Orientación a Objetos (OO). Unido a este paradigma, el denominado *Proceso Unificado* (UP) constituye hoy en día, como procedimiento estandarizado, el marco de trabajo genérico más extendido para el desarrollo de soluciones informáticas, en conjunción con el uso de un lenguaje simbólico común, el *Lenguaje Unificado de Modelado* (UML), para la notación y la comunicación del análisis y el diseño de dichas soluciones.

A partir de aquí, se ha pretendido impulsar el acercamiento de los profesionales e investigadores de la estadística a esta nueva manera de entender el desarrollo de software, haciéndolo además de una manera activa. Para ello, se ha trabajado en la construcción de un *framework* específico para el modelado estadístico con MLG (*véase* apartado 6.2), con la intención de mostrar el papel crucial del UP como guía del desarrollo de este sistema, focalizando los esfuerzos en el análisis del contexto (especificación de los requisitos que debe cumplir) y en el diseño (cómo crear y organizar sus elementos para cumplir con su cometido). Además, los artefactos generados durante el proceso –diagramas de casos de uso, diagramas de clases y diagramas de secuencia, en este caso– han sido publicados en UML, dejando constancia de los resultados en una notación coherente, concisa y precisa.

De hecho, estos artefactos, como productos derivados del proceso, han sido fundamentales en el desarrollo del *framework*, puesto que han permitido contrastar en todo momento si la arquitectura del sistema iba evolucionando de manera adecuada hacia la obtención del producto final. De esta manera, se han reducido riesgos durante el proceso de creación del *framework*, aspecto éste que asegura la optimización de la implementación final del producto en un lenguaje de programación específico, puesto que dicha implementación viene determinada por el diseño.

En este sentido, el salto entre el diseño de la aplicación (plasmado en forma de diagramas de clases y diagramas de secuencia) y su implementación es prácticamente directo, de ahí que se haya dejado en segundo plano este aspecto durante el desarrollo de la parte empírica, centrándonos en el análisis y diseño del *framework*. Ahora bien, con la intención de demostrar que efectivamente este paso del diseño a la implementación es casi directo, se

han redactado ejemplos concretos en código Java durante la exposición del proceso de creación del *framework*, ejemplos que muestran la implementación de clases concretas y del uso de determinadas partes de este sistema. De hecho, de esta manera se constata el carácter iterativo e incremental del UP, reflejado en la estrategia de desarrollo de un producto software en pequeños pasos manejables: planificar un poco; especificar un poco; diseñar un poco; implementar un poco; y vuelta a empezar.

Otra virtud de los artefactos del UP es que producen un diseño no ambiguo de la solución al problema, gracias a que son descritos bajo la especificación UML, aspecto éste que permite precisamente una implementación optimizada de dicha solución. Además, a nivel de investigación, proporcionan un lenguaje estandarizado de comunicación.

En definitiva, se puede decir que los artefactos UML permiten acumular el conocimiento y, además, facilitan la transmisión de dicho conocimiento de manera óptima, puesto que UML ofrece una forma precisa y sintética de abordar la descripción de los productos generados durante el UP. De hecho, creemos que ha quedado suficientemente constatado que el salto entre el dominio del problema (realidad) y el dominio de la solución (implementación) se ha visto reducido gracias a los artefactos UP generados durante el desarrollo de nuestro *framework*.

En este sentido, la propia metodología empleada en la exposición del desarrollo del *framework* destaca también por su capacidad de permitir la acumulación de conocimiento concreto sobre cómo se debe afrontar la búsqueda de soluciones concretas ante problemas específicos. Esto es, el interés principal durante desarrollo del *framework* no ha sido, como decíamos antes, el de plasmar en código la implementación de este sistema, sino que se ha centrado en la justificación permanente de las tomas de decisión relacionadas con la selección de patrones de diseño adecuados para resolver un problema concreto.

Realmente, hubiera sido incongruente hacer énfasis en el análisis y diseño de un sistema, y al mismo tiempo haber focalizado en exceso nuestra atención en el código fuente de dicho sistema. Por ese motivo, el interés real de este trabajo no ha sido la implementación de un sistema concreto –en este caso, un sistema centrado en el contexto del MLG–, sino el análisis de los requisitos asociados a dicho sistema (qué debe hacer) y el correspondiente

diseño de la solución (cómo lo debe hacer). La implementación es en nuestra opinión en este caso un artefacto secundario.

Nótese que la mayoría de trabajos científicos publicados con el objetivo de aportar nuevos productos de software –que representan extensiones de aplicaciones concretas o nuevas aplicaciones en sí mismas–, se centran en describir precisamente las prestaciones de dichos productos, tocando por encima cuestiones de análisis y de diseño, y plasmando un código completo prácticamente inaccesible, puesto que normalmente ha sido codificado en un lenguaje de programación específico habitualmente desconocido por una gran parte de la comunidad científica a la que va dirigida dicha publicación.

Esta manera de plantear la divulgación de un nuevo producto de software deja fuera elementos de gran valor para la comunidad científica, puesto que no se comunica cuáles han sido los criterios importantes en el desarrollo de una aplicación concreta, y por consiguiente, no se establecen las bases suficientes para que la solución aportada pueda ser reutilizada en el desarrollo de nuevas aplicaciones o extensiones.

Pues bien, en este sentido, nuestro trabajo enfatiza el análisis y diseño de la aplicación, con un objetivo claro, transmitir conocimiento sobre las decisiones de diseño que se han adoptado ante un análisis previo de los requisitos de funcionamiento del sistema. En este caso, no sólo los artefactos UP son informativos por sí mismos (diagramas UML del *framework*), sino que además el propio documento aporta información de valor cuando se discute sobre la idoneidad de aplicar uno u otro diseño ante un determinado problema. Además, no se discute sobre diseños improvisados en el momento, sino sobre diseños conocidos, diseños estándar y genéricos que aportan soluciones concretas ante problemas recurrentes. Esto es, se discute sobre la aplicación de los denominados *patrones de diseño*, cuyo uso ha sido clave en el desarrollo de un diseño óptimo del *framework*.

Precisamente, el análisis de un problema dado y la búsqueda de soluciones concretas en forma de patrones de diseño, y el hecho de que la implementación de este diseño sea prácticamente directa bajo un entorno orientado a objetos, permite allanar el camino para que un profesional de la estadística sea capaz de desarrollar software estadístico.

Además, las ideas de reusabilidad y fácil extensión de sistemas, plasmadas de manera óptima y eficiente en nuestro *framework*, ofrecen también un valor añadido en el contexto

de la programación estadística; aportan facilidades al profesional que está interesado en ampliar un determinado sistema, incorporando una nueva clase con una mínima inversión de tiempo de programación, ya que aprovecha toda la estructura de clases de dicho sistema para añadir la funcionalidad de interés.

En este sentido, la facilidad de los lenguajes orientados a objetos para ampliar sistemas de manera segura y relativamente sencilla, puede verse además reforzada por el acceso a entornos de desarrollo integrados (EDI), los cuales permiten abstraerse, en gran parte, de las cuestiones relacionadas con el lenguaje de programación concreto utilizado para la implementación.

En relación al *framework* desarrollado, si bien está preparado tanto para el modelado estadístico como para la simulación estocástica en el contexto del MLG, un planteamiento de futuro importante pasa por comprobar si efectivamente este diseño es estable y extensible. Para ello, se podría tratar de extender el sistema más allá del marco MLG, o mejor dicho, del marco de los modelos estadísticos del ámbito de la familia exponencial de distribuciones, ya que dentro de este ámbito se ha intentado presentar la vía de extensibilidad planteando someramente las derivaciones que permitirían la inclusión de los Modelos Aditivos Generalizados (GAM) en nuestro *framework*.

Además, retomando la idea de que el diseño de este sistema es independiente de la interfaz utilizada para implementar su funcionalidad, sería interesante sumergir dicho diseño en un EDI capaz de guiar al usuario en la extensión de dicho sistema, aspecto éste que nos planteamos como una de los principales líneas de trabajo derivadas de la tesis.

Un elemento fundamental a destacar en relación a los diagramas de clases asociados al diseño del *framework* desarrollado (*véase* anexo), es que éstos son mucho más informativos de lo que aparentan a simple vista. En este sentido, detallan información no ambigua, e indican los «estereotipos» concretos de una determinada clase y los patrones de diseño que se aplican a un grupo concreto de clases. Queda clara, por tanto, la riqueza del lenguaje UML para acumular conocimiento preciso sobre el diseño de un sistema.

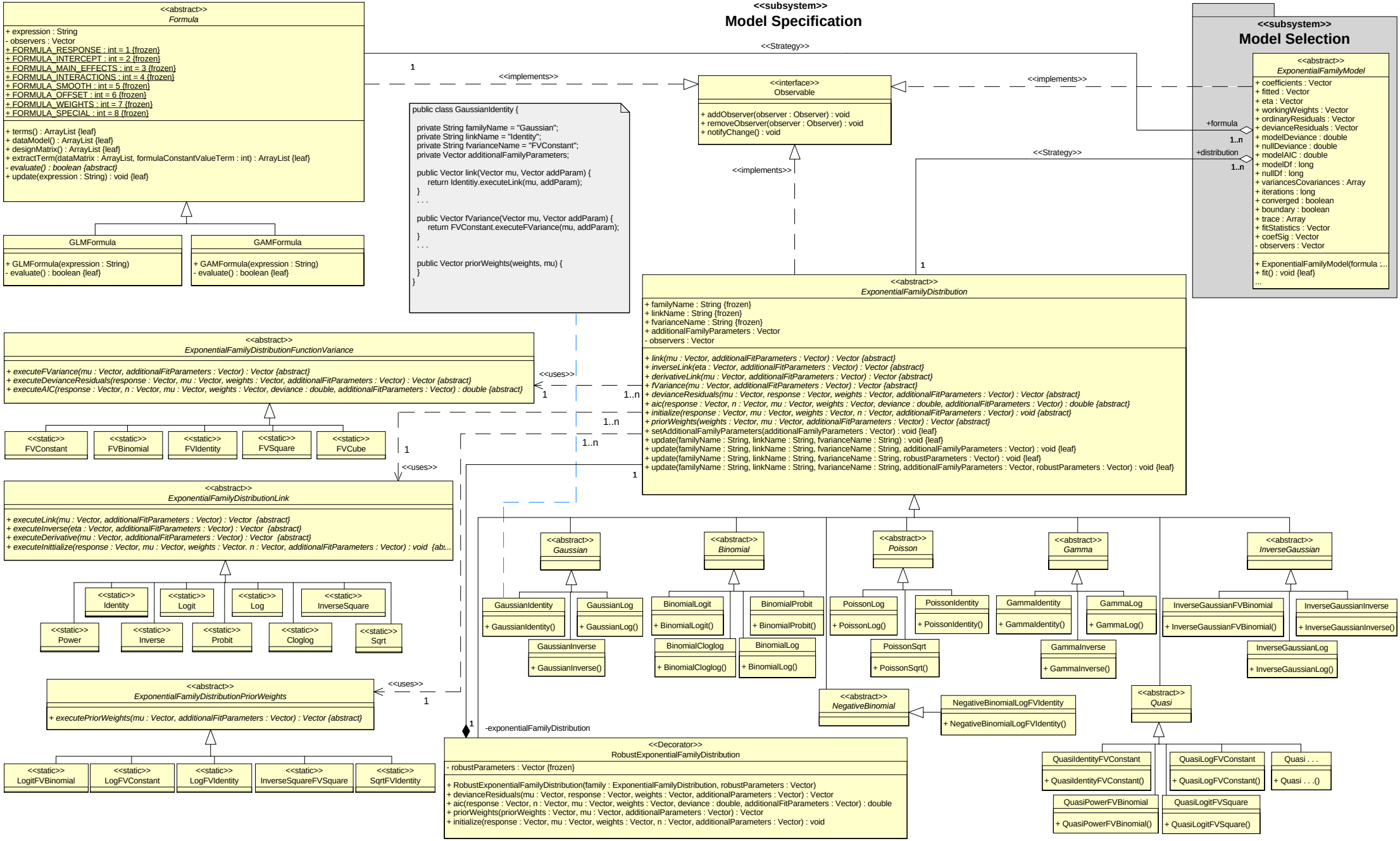
Por último, se plantean una serie de reflexiones sobre las perspectivas de futuro del UP combinado con UML, y enmarcados ambos en el paradigma de la OO.

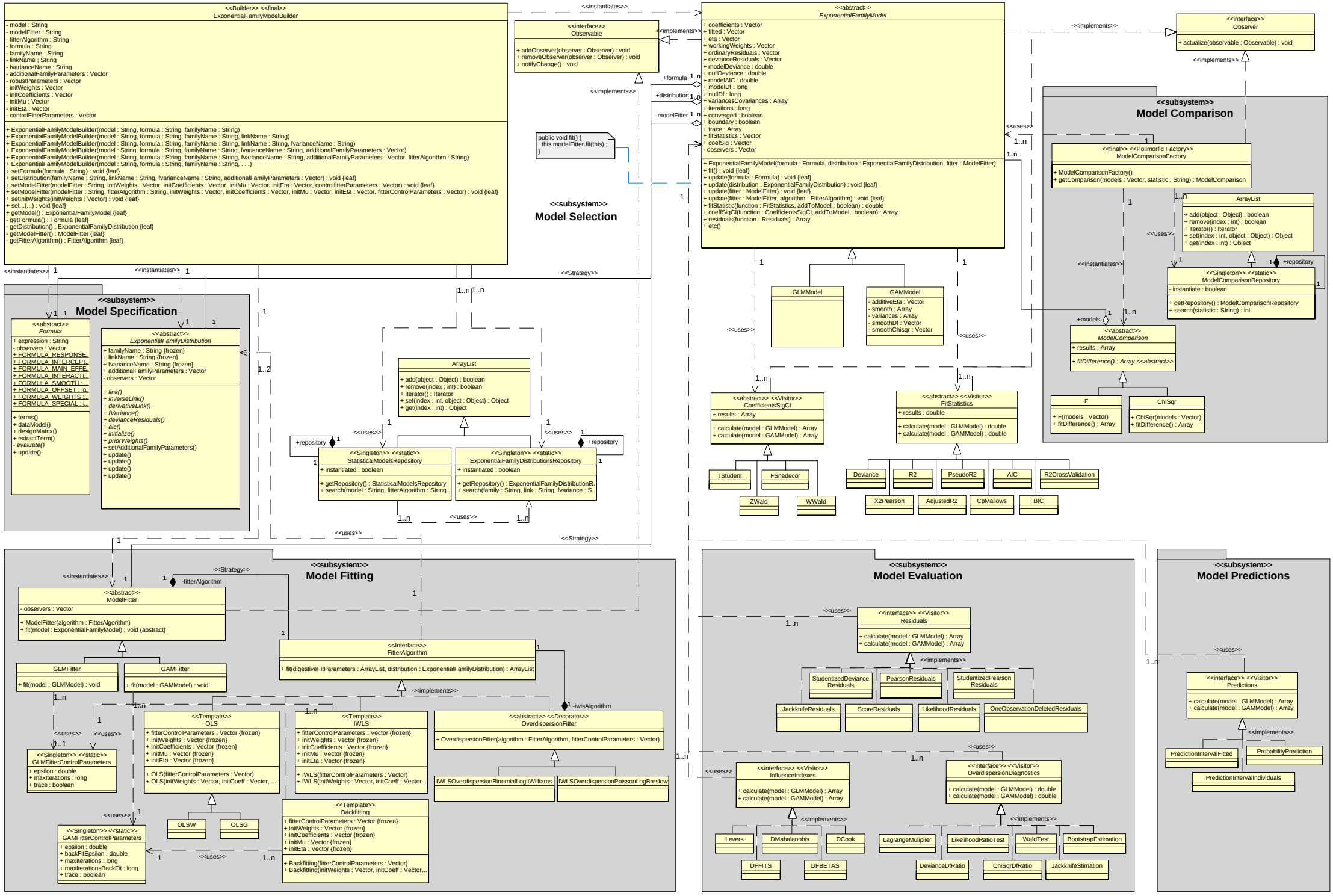
En primer lugar, hay que destacar que esta combinación de elementos permite obtener una serie de artefactos que constituyen una herramienta de comunicación científica de primera magnitud.

Además, estas herramientas metodológicas podrían trascender a otros contextos, con la finalidad de aportar procedimientos formales que permitan el desarrollo de artefactos propios y que al mismo tiempo sean descritos con un alto nivel de detalle, con la intención de permitir una acumulación de conocimiento realmente fructífera. En este sentido, podría ser interesante exportar dichas herramientas al contexto del modelado de teorías psicológicas. El uso de herramientas de comunicación formales y estandarizadas, como la especificación UML, permitiría un claro avance en la mejora de la comunicación en este contexto, produciéndose a su vez una adecuada transmisión de la información, con un nivel de detalle suficiente y preciso.

Por otra parte, también podría ser interesante utilizar dichas herramientas para facilitar el aprendizaje didáctico en el campo educativo. Partiendo de un proceso estandarizado como el UP, se tendría acceso de manera progresiva a los diferentes artefactos necesarios para la asimilación de los contenidos de una materia concreta; en este sentido, cada uno de estos artefactos presentaría una nivel de abstracción distinto, del mismo modo en que los diagramas de casos de uso de un sistema presentan un menor nivel de detalle que los diagramas de clases del diseño.

ANEXO


<<subsystem>>
Model Specification
<<subsystem>>
Model Selection
<<abstract>>
Formula
+ expression : String
- observers : Vector
+ FORMULA_RESPONSE : int = 1 {frozen}
+ FORMULA_INTERCEPT : int = 2 {frozen}
+ FORMULA_MAIN_EFFECTS : int = 3 {frozen}
+ FORMULA_INTERACTIONS : int = 4 {frozen}
+ FORMULA_SMOOTH : int = 5 {frozen}
+ FORMULA_OFFSET : int = 6 {frozen}
+ FORMULA_WEIGHTS : int = 7 {frozen}
+ FORMULA_SPECIAL : int = 8 {frozen}
+ terms() : ArrayList {leaf}
+ dataModel() : ArrayList {leaf}
+ designMatrix() : ArrayList {leaf}
+ extractTerm(dataMatrix : ArrayList, formulaConstantValueTerm : int) : ArrayList {leaf}
- evaluate() : boolean {abstract}
+ update(expression : String) : void {leaf}
GLMFormula
+ GLMFormula(expression : String)
- evaluate() : boolean {leaf}
GAMFormula
+ GAMFormula(expression : String)
- evaluate() : boolean {leaf}
<<Strategy>>
1
<<implements>>
<<interface>>
Observable
+ addObserver(observer : Observer) : void
+ removeObserver(observer : Observer) : void
+ notifyChange() : void
<<implements>>
<<abstract>>
ExponentialFamilyModel
+ coefficients : Vector
+ fitted : Vector
+ eta : Vector
+ workingWeights : Vector
+ ordinaryResiduals : Vector
+ devianceResiduals : Vector
+ modelDeviance : double
+ nullDeviance : double
+ modelAIC : double
+ modelDf : long
+ nullDf : long
+ variancesCovariances : Array
+ iterations : long
+ converged : boolean
+ boundary : boolean
+ trace : Array
+ fitStatistics : Vector
+ coefSig : Vector
- observers : Vector
+ ExponentialFamilyModel(formula :...
+ fit() : void {leaf}
...
+formula
1..n
+distribution
1..n
<<Strategy>>
public class GaussianIdentity {
private String familyName = "Gaussian";
private String linkName = "Identity";
private String fvarianceName = "FVConstant";
private Vector additionalFamilyParameters;
public Vector link(Vector mu, Vector addParam) {
return Identitiy.executeLink(mu, addParam);
}
. . .
public Vector fVariance(Vector mu, Vector addParam) {
return FVConstant.executeFVariance(mu, addParam);
}
. . .
public Vector priorWeights(weights, mu) {
}
}
<<abstract>>
ExponentialFamilyDistribution
+ familyName : String {frozen}
+ linkName : String {frozen}
+ fvarianceName : String {frozen}
+ additionalFamilyParameters : Vector
- observers : Vector
+ link(mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ inverseLink(eta : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ derivativeLink(mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ fVariance(mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ devianceResiduals(mu : Vector, response : Vector, weights : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ aic(response : Vector, n : Vector, mu : Vector, weights : Vector, deviance : double, additionalFitParameters : Vector) : double {abstract}
+ initialize(response : Vector, mu : Vector, weights : Vector, n : Vector, additionalFitParameters : Vector) : void {abstract}
+ priorWeights(weights : Vector, mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ setAdditionalFamilyParameters(additionalFamilyParameters : Vector) : void {leaf}
+ update(familyName : String, linkName : String, fvarianceName : String) : void {leaf}
+ update(familyName : String, linkName : String, fvarianceName : String, additionalFamilyParameters : Vector) : void {leaf}
+ update(familyName : String, linkName : String, fvarianceName : String, robustParameters : Vector) : void {leaf}
+ update(familyName : String, linkName : String, fvarianceName : String, additionalFamilyParameters : Vector, robustParameters : Vector) : void {leaf}
<<abstract>>
ExponentialFamilyDistributionFunctionVariance
+ executeFVariance(mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ executeDevianceResiduals(response : Vector, mu : Vector, weights : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ executeAIC(response : Vector, n : Vector, mu : Vector, weights : Vector, deviance : double, additionalFitParameters : Vector) : double {abstract}
<<uses>>
1
1..n
<<static>>
FVConstant
<<static>>
FVBinomial
<<static>>
FVIdentity
<<static>>
FVSquare
<<static>>
FVCube
<<abstract>>
ExponentialFamilyDistributionLink
+ executeLink(mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ executeInverse(eta : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ executeDerivative(mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
+ executeInittialize(response : Vector, mu : Vector, weights : Vector. n : Vector, additionalFitParameters : Vector) : void {abs...
<<static>>
Identity
<<static>>
Logit
<<static>>
Log
<<static>>
InverseSquare
<<static>>
Power
<<static>>
Inverse
<<static>>
Probit
<<static>>
Cloglog
<<static>>
Sqrt
<<abstract>>
Gaussian
GaussianIdentity
+ GaussianIdentity()
GaussianLog
+ GaussianLog()
GaussianInverse
+ GaussianInverse()
<<abstract>>
Binomial
BinomialLogit
+ BinomialLogit()
BinomialProbit
+ BinomialProbit()
BinomialCloglog
+ BinomialCloglog()
BinomialLog
+ BinomialLog()
<<abstract>>
Poisson
PoissonLog
+ PoissonLog()
PoissonIdentity
+ PoissonIdentity()
PoissonSqrt
+ PoissonSqrt()
<<abstract>>
Gamma
GammaIdentity
+ GammaIdentity()
GammaLog
+ GammaLog()
GammaInverse
+ GammaInverse()
<<abstract>>
InverseGaussian
InverseGaussianFVBinomial
+ InverseGaussianFVBinomial()
InverseGaussianInverse
+ InverseGaussianInverse()
InverseGaussianLog
+ InverseGaussianLog()
<<abstract>>
ExponentialFamilyDistributionPriorWeights
+ executePriorWeights(mu : Vector, additionalFitParameters : Vector) : Vector {abstract}
<<static>>
LogitFVBinomial
<<static>>
LogFVConstant
<<static>>
LogFVIdentity
<<static>>
InverseSquareFVSquare
<<static>>
SqrtFVIdentity
-exponentialFamilyDistribution
<<Decorator>>
RobustExponentialFamilyDistribution
- robustParameters : Vector {frozen}
+ RobustExponentialFamilyDistribution(family : ExponentialFamilyDistribution, robustParameters : Vector)
+ devianceResiduals(mu : Vector, response : Vector, weights : Vector, additionalParameters : Vector) : Vector
+ aic(response : Vector, n : Vector, mu : Vector, weights : Vector, deviance : double, additionalFitParameters : Vector) : double
+ priorWeights(priorWeights : Vector, mu : Vector, additionalParameters : Vector) : Vector
+ initialize(response : Vector, mu : Vector, weights : Vector, n : Vector, additionalParameters : Vector) : void
<<abstract>>
NegativeBinomial
NegativeBinomialLogFVIdentity
+ NegativeBinomialLogFVIdentity()
<<abstract>>
Quasi
QuasiIdentityFVConstant
+ QuasiIdentityFVConstant()
QuasiLogFVConstant
+ QuasiLogFVConstant()
Quasi . . .
+ Quasi . . .()
QuasiPowerFVBinomial
+ QuasiPowerFVBinomial()
QuasiLogitFVSquare
+ QuasiLogitFVSquare()


<<Builder>> <<final>>
ExponentialFamilyModelBuilder
- model : String
- modelFitter : String
- fitterAlgorithm : String
- formula : String
- familyName : String
- linkName : String
- fvarianceName : String
- additionalFamilyParameters : Vector
- robustParameters : Vector
- initWeights : Vector
- initCoefficients : Vector
- initMu : Vector
- initEta : Vector
- controlFitterParameters : Vector
+ ExponentialFamilyModelBuilder(model : String, formula : String, familyName : String)
+ ExponentialFamilyModelBuilder(model : String, formula : String, familyName : String, linkName : String)
+ ExponentialFamilyModelBuilder(model : String, formula : String, familyName : String, linkName : String, fvarianceName : String)
+ ExponentialFamilyModelBuilder(model : String, formula : String, familyName : String, fvarianceName : String, additionalFamilyParameters : Vector)
+ ExponentialFamilyModelBuilder(model : String, formula : String, familyName : String, fvarianceName : String, additionalFamilyParameters : Vector, fitterAlgorithm : String)
+ ExponentialFamilyModelBuilder(model : String, formula : String, familyName : String, . . .)
+ setFormula(formula : String) : void {leaf}
+ setDistribution(familyName : String, linkName : String, fvarianceName : String, additionalFamilyParameters : Vector) : void {leaf}
+ setModelFitter(modelFitter : String, initWeights : Vector, initCoefficients : Vector, initMu : Vector, initEta : Vector, controlfitterParameters : Vector) : void {leaf}
+ setModelFitter(modelFitter : String, fitterAlgorithm : String, initWeights : Vector, initCoefficients : Vector, initMu : Vector, initEta : Vector, fitterControlParameters : Vector) : void {leaf}
+ setInitWeights(initWeights : Vector) : void {leaf}
+ set...(...) : void {leaf}
+ getModel() : ExponentialFamilyModel {leaf}
- getFormula() : Formula {leaf}
- getDistribution() : ExponentialFamilyDistribution {leaf}
- getModelFitter() : ModelFitter {leaf}
- getFitterAlgorithm() : FitterAlgorithm {leaf}
<<instantiates>>
<<abstract>>
ExponentialFamilyModel
+ coefficients : Vector
+ fitted : Vector
+ eta : Vector
+ workingWeights : Vector
+ ordinaryResiduals : Vector
+ devianceResiduals : Vector
+ modelDeviance : double
+ nullDeviance : double
+ modelAIC : double
+ modelDf : long
+ nullDf : long
+ variancesCovariances : Array
+ iterations : long
+ converged : boolean
+ boundary : boolean
+ trace : Array
+ fitStatistics : Vector
+ coefSig : Vector
- observers : Vector
+ ExponentialFamilyModel(formula : Formula, distribution : ExponentialFamilyDistribution, fitter : ModelFitter)
+ fit() : void {leaf}
+ update(formula : Formula) : void {leaf}
+ update(distribution : ExponentialFamilyDistribution) : void {leaf}
+ update(fitter : ModelFitter) : void {leaf}
+ update(fitter : ModelFitter, algorithm : FitterAlgorithm) : void {leaf}
+ fitStatistic(function : FitStatistics, addToModel : boolean) : double
+ coeffSigCI(function : CoefficientsSigCI, addToModel : boolean) : Array
+ residuals(function : Residuals) : Array
+ etc()
<<interface>>
Observable
+ addObserver(observer : Observer) : void
+ removeObserver(observer : Observer) : void
+ notifyChange() : void
<<implements>>
<<interface>>
Observer
+ actualize(observable : Observable) : void
+formula
+distribution
-modelFitter
1..n
public void fit() {
this.modelFitter.fit(this) ;
}
<<uses>>
<<subsystem>>
Model Selection
<<subsystem>>
Model Comparison
<<final>> <<Polimorfic Factory>>
ModelComparisonFactory
+ ModelComparisonFactory()
+ getComparison(models : Vector, statistic : String) : ModelComparison
ArrayList
+ add(object : Object) : boolean
+ remove(index ; int) : boolean
+ iterator() : Iterator
+ set(index : int, object : Object) : Object
+ get(index : int) : Object
<<Singleton>> <<static>>
ModelComparisonRepository
- instantiate : boolean
+ getRepository() : ModelComparisonRepository
+ search(statistic : String) : int
+repository
+models
<<abstract>>
ModelComparison
+ results : Array
+ fitDifference() : Array <<abstract>>
F
+ F(models : Vector)
+ fitDifference() : Array
ChiSqr
+ ChiSqr(models : Vector)
+ fitDifference() : Array
GLMModel
GAMModel
- additiveEta : Vector
- smooth : Array
- variances : Array
- smoothDf : Vector
- smoothChisqr : Vector
<<abstract>> <<Visitor>>
CoefficientsSigCI
+ results : Array
+ calculate(model : GLMModel) : Array
+ calculate(model : GAMModel) : Array
<<abstract>> <<Visitor>>
FitStatistics
+ results : double
+ calculate(model : GLMModel) : double
+ calculate(model : GAMModel) : double
TStudent
FSnedecor
ZWald
WWald
Deviance
R2
PseudoR2
AIC
R2CrossValidation
X2Pearson
AdjustedR2
CpMallows
BIC
<<subsystem>>
Model Specification
<<abstract>>
Formula
+ expression : String
- observers : Vector
+ FORMULA_RESPONSE..
+ FORMULA_INTERCEPT..
+ FORMULA_MAIN_EFFE..
+ FORMULA_INTERACTI...
+ FORMULA_SMOOTH : ...
+ FORMULA_OFFSET : ig..
+ FORMULA_WEIGHTS : ...
+ FORMULA_SPECIAL : i..
+ terms()
+ dataModel()
+ designMatrix()
+ extractTerm()
- evaluate()
+ update()
<<abstract>>
ExponentialFamilyDistribution
+ familyName : String {frozen}
+ linkName : String {frozen}
+ fvarianceName : String {frozen}
+ additionalFamilyParameters : Vector
- observers : Vector
+ link()
+ inverseLink()
+ derivativeLink()
+ fVariance()
+ devianceResiduals()
+ aic()
+ initialize()
+ priorWeights()
+ setAdditionalFamilyParameters()
+ update()
+ update()
+ update()
+ update()
ArrayList
+ add(object : Object) : boolean
+ remove(index ; int) : boolean
+ iterator() : Iterator
+ set(index : int, object : Object) : Object
+ get(index : int) : Object
<<Singleton>> <<static>>
StatisticalModelsRepository
+ instantiated : boolean
+ getRepository() : StatisticalModelsRepository
+ search(model : String, fitterAlgorithm : String..
<<Singleton>> <<static>>
ExponentialFamilyDistributionsRepository
+ instantiated : boolean
+ getRepository() : ExponentialFamilyDistributionR..
+ search(family : String, link : String, fvariance : S..
+repository
<<Strategy>>
<<subsystem>>
Model Fitting
<<abstract>>
ModelFitter
- observers : Vector
+ ModelFitter(algorithm : FitterAlgorithm)
+ fit(model : ExponentialFamilyModel) : void {abstract}
-fitterAlgorithm
<<Interface>>
FitterAlgorithm
+ fit(digestiveFitParameters : ArrayList, distribution : ExponentialFamilyDistribution) : ArrayList
GLMFitter
+ fit(model : GLMModel) : void
GAMFitter
+ fit(model : GAMModel) : void
<<Template>>
OLS
+ fitterControlParameters : Vector {frozen}
+ initWeights : Vector {frozen}
+ initCoefficients : Vector {frozen}
+ initMu : Vector {frozen}
+ initEta : Vector {frozen}
+ OLS(fitterControlParameters : Vector)
+ OLS(initWeights : Vector, initCoeff : Vector, ...
<<Template>>
IWLS
+ fitterControlParameters : Vector {frozen}
+ initWeights : Vector {frozen}
+ initCoefficients : Vector {frozen}
+ initMu : Vector {frozen}
+ initEta : Vector {frozen}
+ IWLS(fitterControlParameters : Vector)
+ IWLS(initWeights : Vector, initCoeff : Vector...
-iwlsAlgorithm
<<abstract>> <<Decorator>>
OverdispersionFitter
+ OverdispersionFitter(algorithm : FitterAlgorithm, fitterControlParameters : Vector)
IWLSOverdispersionBinomialLogitWilliams
IWLSOverdispersionPoissonLogBreslow
<<Singleton>> <<static>>
GLMFitterControlParameters
+ epsilon : double
+ maxIterations : long
+ trace : boolean
<<Singleton>> <<static>>
GAMFitterControlParameters
+ epsilon : double
+ backFitEpsilon : double
+ maxIterations : long
+ maxIterationsBackFit : long
+ trace : boolean
OLSW
OLSG
<<Template>>
Backfitting
+ fitterControlParameters : Vector {frozen}
+ initWeights : Vector {frozen}
+ initCoefficients : Vector {frozen}
+ initMu : Vector {frozen}
+ initEta : Vector {frozen}
+ Backfitting(fitterControlParameters : Vector)
+ Backfitting(initWeights : Vector, initCoeff : Vector...
<<subsystem>>
Model Evaluation
<<interface>> <<Visitor>>
Residuals
+ calculate(model : GLMModel) : Array
+ calculate(model : GAMModel) : Array
StudentizedDeviance Residuals
PearsonResiduals
StudentizedPearson Residuals
JackknifeResiduals
ScoreResiduals
LikelihoodResiduals
OneObservationDeletedResiduals
<<interface>> <<Visitor>>
InfluenceIndexes
+ calculate(model : GLMModel) : Array
+ calculate(model : GAMModel) : Array
<<interface>> <<Visitor>>
OverdispersionDiagnostics
+ calculate(model : GLMModel) : double
+ calculate(model : GAMModel) : double
Levers
DMahalanobis
DCook
DFFITS
DFBETAS
LagrangeMuliplier
LikelihoodRatioTest
WaldTest
BootstrapEstimation
DevianceDfRatio
ChiSqrDfRatio
JackknifeStimation
<<subsystem>>
Model Predictions
<<interface>> <<Visitor>>
Predictions
+ calculate(model : GLMModel) : Array
+ calculate(model : GAMModel) : Array
PredictionIntervalFitted
ProbabilityPrediction
PredictionIntervalIndividuals

REFERENCIAS

Abbott, R.J. (1983). Program design by informal English descriptions. *Communications ACM, 26*(11), 882-894.

Alfonseca, M. y Alcalá, A. (1992). *Programación orientada a objetos. Teoría y técnicas OOP para desarrollo de software*. Madrid: Ediciones Anaya Multimedia, S.A.

Anderson, D.E. (1995). Extensible programming: Beyond reusable objects. *Behavior Research Methods, Instruments & Computers, 27*(2), 131-133.

Arango, G. (1989). Domain Analysis: From Art Form to Engineering Discipline. *SIGSOFT Engineering Notes, 14*(3), 152-159.

Arnau, J. (1989). Metodología de la investigación y diseño. En J. Arnau y J. Carpintero (Eds.). *Tratado de psicología general I: Historia, Teoría y Método* (pp. 581-616). Madrid: Alhambra.

Beck, K. y Cunningham, W. (1989). A Laboratory for Teaching Object-Oriented Thinking. *SIGPLAN Notices, 24*(10), 1-6.

Blum, A. (1992). *Neural Networks in C++: An Object-Oriented Framework for Building Connectionist Systems*. New York: Wiley.

Booch, G. (1986). *Object-Oriented Development. IEEE Transactions on Software Engineering, 12*(2), 211-221.

Booch, G. (1991). *Object Oriented Design with Applications*. Redwood City, CA: Benjamin Cummings Publishing Company.

Booch, G. (1996). *Análisis y diseño orientado a objetos con aplicaciones* (2ª ed.). (J.M. Cueva y A. Cernuda, Trads.). Wilmington, Delaware: Addison-Wesley Iberoamericana (Original en inglés publicado en 1994).

Booch, G., Rumbaugh, J. y Jacobson, I. (1999). *El Lenguaje Unificado de Modelado*. (J. Sáez, Trad.). Madrid: Addison Wesley Iberoamericana (Original en inglés publicado en 1998).

Brooks, F.P. (1975). *The Mytical Man-month*. Reading, MA: Addison-Wesley.

Cameron, A.C. y Trivedi, P.K. (1998). *Regression analysis of count data*. Cambridge: Cambridge University Press.

Cardelli, L. y Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys, 17*(4), 471-522.

Chambers, J.M. (1988). *Programming with Data. A Guide to the S Language*. New York: Springer-Verlag.

Chambers, J.M. (2000). *Users, programmers, and statistical software. Journal of Computational and Graphical Statistics, 9*(3), 404-422.

Chambers, J.M. y Hastie, T.J. (1991). *Statistical Models in S*. Pacific Grove, CA: Wadswort & Brooks/Cole.

Coad, P. (1992). Object-oriented patterns. *Communications ACM, 35*(9), 152-158.

Coad, P. y Yourdon. E. (1991a). *Object-Oriented Analysis* (2ª ed.). Englewood Cliffs, NJ: Yourdon Press/Prentice Hall.

Coad, P. y Yourdon. E. (1991b). *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdon Press/Prentice Hall.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. et al. (1994). *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, NJ: Prentice Hall.

Dahl, O., Dijkstra, E. y Hoare, C.A.R. (1972). *Structured Programming*. London: Academic Press.

Dahl, O.J. y Nygaard, K. (1996). SIMULA – An Algol-based simulation language. *Communications ACM, 9,* 671-678.

Dahl, O.J., Myrhaug, B. y Nygaard, K. (1968). *SIMULA 67. Common Base Language*. Oslo: Norwegian Computing Centre.

de Champeaux, D. y Faure, P. (1992). A comparative study of Object-Oriented Analysis Methods. *Journal of Object-Oriented Programming, 5*(1), 21-33.

De Marco, T. (1982). *Controlling Software Projects*. Englewood Cliffs, NJ: Prentice Hall.

Desfray, P. (1992). *Ingénerie des objetcts: Approche classe-relation application à C++*. Paris: Editions Masson.

Eckel, B. (2003). *Thinking in Patterns with Java*. Libro en preparación, la versión preliminar es accesible por Internet: http://www.mindview.net/Books/TIPatterns.

Embley, D.W., Kurtz, B.D. y Woodfield, S.N. (1992). *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs, NJ: Yourdon Press.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (2003). *Patrones de diseño. Elementos de software orientado a objetos reutilizable*. (C. Fernández, Trad.). Madrid: Pearson Educación, S.A. (Original en inglés publicado en 1995).

Gilb, T. (1990). *Principles of Software Project Management* (2ª ed.). New York: Addison-Wesley.

Gill, J. (2000). *Generalized Linear Models: A Unified Approach*. Thousand Oaks, CA: Sage.

Gosling, J., Joy, B. y Steele, G. (1996). *The Java Language Specification*. Reading, MA: Addison-Wesley.

Graham, I. (1994a). *Migrating to Object Technology*. Workingham: Addison-Wesley.

Graham, I. (1994b). *Object Oriented Methods* (2ª ed.). Workingham: Addison-Wesley.

Grupo ModEst. (2000a). *Del Contraste de Hipótesis al Modelado Estadístico*. Terrassa: CBS.

Grupo ModEst. (2000b). *Modelo Lineal Generalizado*. Terrassa: CBS.

Hardin, J. y Hilbe, J. (2001). *Generalized Linear Models and Extensions*. Texas: Stata Press.

Hastie, T. y Tibshirani, R. (1990). *Generalized additive models*. London: Chapman & Hall.

Hitz, M. y Hudec, M. (1994). Applying the Object Oriented Paradigm to Statistical Computing. *Proceedings in Computational Statistics, COMPSTAT 94*, 389-394.

Hutcheson, G.D. y Sofroniou, N. (1999). *The Multivariate Social Scientist. Introductory Statistics Using Generalized Linear Models*. London: Sage.

Ihaka, R. y Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics, 5*(3), 299-314.

Jacobson, I. (1995). Formalizing Use-Case Modeling. *Journal of Object-Oriented Programming, 8*(3), 10-14.

Jacobson, I. y Christerson, M. (1995). *A growing consensus on use cases. Journal of Object-Oriented Programming, 8*(1), 15-19.

Jacobson, I., Booch, G. y Rumbaugh, J. (2000). *El Proceso Unificado de Desarrollo de Software*. (S. Sánchez, M.A. Sicilia, C. Canal y F.J. Durán, Trads.). Madrid: Pearson Educación, S.A. (Original en inglés publicado en 1999).

Jacobson, I., Christerson, M., Jonsson, P. y Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Workingham: Addison-Wesley.

Johnson, E. y Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming, 1*(2), 22-35.

Joyanes, L. y Zahonero, I. (2002). *Programación en Java 2. Algoritmos, Estructuras de Datos y Programación Orientada a Objetos*. Madrid: McGraw-Hill.

Judd, C.M. y McClelland, G.H. (1989). *Data Analysis: A Model-Comparison Approach*. San Diego, CA: University Press.

Krzanowski, W.J. (1998). *An Introduction to Statistical Modelling*. London: Arnold.

Larman, C. (2003). *UML y PATRONES. Una introducción al análisis y diseño orientado a objetos y al proceso unificado* (2ª ed.). (Begoña, M., Trad.). Madrid: Pearson Educación, S.A. (Original en inglés publicado en 2002).

Lindsey, J.K. (1997). *Applying Generalized Linear Models*. New York: Springer-Verlag.

Long, J.S. (1997). *Regression models for categorical and limited dependent variables*. Thousand Oaks, CA: Sage.

Losilla, J.M. (1995). *Proyecto docente de la asignatura «Software» en Psicología* (Manuscrito no publicado). Barcelona: Universitat Autònoma de Barcelona, Departament de Psicologia de la Salut.

Losilla, J.M. (2003). *El Proceso Unificado de Desarrollo de Software Estadístico*. Versión previa policopiada de la comunicación a presentar en el Simposio "Desarrollo de Software Estadístico: metodologías, diseño y entornos de implementación" del VIII Congreso de Metodología de las CC. Sociales y de la Salud a celebrar en Valencia en Septiembre de 2003.

Mallows, C.L. (1973). Some comments on Cp. *Technometrics, 15*, 661-675.

Martín, J. y Odell, J.J. (1994). *Análisis y Diseño Orientado a Objetos*. México: Prentice Hall Hispanoamericana, S.A.

McCullagh, P. y Nelder, J.A. (1983). *Generalized linear models*. London: Chapman & Hall.

McCullagh, P. y Nelder, J.A. (1989). *Generalized linear models* (2ª ed.). London: Chapman & Hall.

McGregor, J.D. y Korson, T. (1990). Understanding Object-Oriented: A Unifying Paradigm. *Communications ACM, 33*(9), 123-136.

Meyer, B. (1991). *Eiffel, The Language*. Englewood Cliffs, NJ: Prentice Hall, Inc.

Moon, D. (1989). The Common LISP object-oriented programming language standard. En Kim, W. y Lochovsky, F.H. (Eds.), *Object-Oriented Concepts, Databases and Applications*. Reading, MA: Addison-Wesley.

Moore, J. y Bailin, S. (1988). *Position Paper on Domain Analysis*. Laurel, MD: CTA.

Moto-Oka, T. y Kitsuregawa, M. (1986). *El ordenador de Quinta Generación*. Barcelona: Ariel.

Nelder, J.A., y Wedderburn, W.M. (1972). Generalized linear models. *Journal of the Royal Statistical Society – Series A, 135*, 370-384.

Ocaña, J. y Sánchez, A. (2003). *Diseño orientado a objetos y software estadístico: patrones de diseño, UML y composición vs. herencia*. Barcelona: documento policopiado. (Puede obtenerse copia enviando petición a los autores al Departament d'Estadística de la Universitat de Barcelona).

OMG. (2003). *OMG Unified Modeling Language Specification*. Recuperado el 18 de marzo de 2003 de http://www.omg.org/technology/documents/formal/uml.htm.

Peralta, A.J. y Rodríguez, H. (1994). *Enginyeria del software. Programació orientada a objectes*. Barcelona: Edicions UPC.

Press, W.H., Flannery, B.P., Teukolsky, S.A. y Vetterling, W.T. (1989). *Numerical Recipes. The art of scientific computing (Fortran Version)*. Cambridge: Cambridge University Press.

Pressman, R.S. (1993). *Ingeniería del Software. Un enfoque práctico* (3ª ed.). Madrid: McGraw-Hill/Interamericana de España, S.A.

Robinson, P. (Ed.). (1992). *Object-Oriented Design*. London: Chapman & Hall.

Rodríguez, G. (2002). *Lecture notes on generalized linear models*. Recuperado el 14 de mayo de 2003 de http://data.princenton.edu/wws509/notes.

Ross, R. (1987). *Entity Modeling: Techniques and Applications*. Boston, MA: Database Research Group.

Rumbaugh, J., Blaha M., Premerlani, W., Eddy, P. y Lorensen, J. (1991). *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice Hall.

Sánchez, A. (1996). *Estudi d'alguns problemes estadístics de la Distància Genètica de Prevosti. Modelització Orientada a Objectes en Estadística i Simulació*. Tesis Doctoral. Barcelona: Universitat de Barcelona - Departament d'Estadística.

Savic, D. (1990). *Object-Oriented programming with Smalltalk-V*. Englewood Cliffs, NJ: Prentice Hall, Inc.

Shlaer, S. y Mellor, S.J. (1988). *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press.

Shlaer, S. y Mellor, S.J. (1991). *Object-Lifecycles: Modelling the World in States*. Englewood Cliffs, NJ: Yourdon Press.

Simons. G.L. (1985). *Los ordenadores de la quinta generación*. Madrid: Ediciones Díaz de Santos, S.A.

Steel, G.L. Jr. (1990). *Common Lisp, The Language*. Reading, MA: Addison-Wesley.

Stelting, S. y Maassen, O. (2001). *Applied Java Patterns*. New York: Prentice Hall.

Stroustrup, B. (1986). *The C++ Programming Language*. Reading, MA: Addison-Wesley Publishing Company, Inc. (2ª edición en español: Wilmington, Delaware: Addison-Wesley Iberoamericana, S.A., 1993).

Stroustrup, B. (1988). What is Object-Oriented programming?. *IEEE Software, 5*(3), 10-20.

Taylor, D. (1992). *Object-Oriented Technology: A Manager's Guide*. Reading, MA: Addison-Wesley.

Temple, L. (2000). The Omegahat Environment: New Possibilities for Statistical Computing. *Journal of Computational and Graphical Statistics, 9*(3), 423-451.

Vives, J. (2002). *El diagnóstico de la sobredispersión en modelos de análisis de datos de recuento*. Tesis Doctoral. Barcelona: Universitat Autònoma de Barcelona - Departament de Psicobiologia i de Metodologia de les Ciències de la Salut.

Wasserman, A.I., Pircher, P.A. y Muller, R.J. (1990). The object-oriented structured design notation for software design representation. *IEEE Computer*, Marzo, 50-62.

Wirfs-Brock, R., Wilkerson, B. y Wiener, L. (1990). *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.